命令/操作レベルの並列処理機能を持つプロセッサ・アーキテクチャの研究(Dissertation_全文)

AUTHOR(S):

北村, 俊明

(2)

命令/操作レベルの並列処理機能を持つプロセッサ・アーキテクチャの研究

北 村 俊 明

1995年12月

目 次

図目次

表目次

Chapter 1

序論

計算機システムアーキテクチャは、計算機が発明されて以来の研究テーマである。現在まで、その時々で、各種のシステムアーキテクチャが提案されてきたが、使用できる論理素子・実装技術・入出力装置や外部記憶装置等の進歩により、常に変化してきた。また、ハードウェア面での進歩以外でも、ソフトウェア、特にコンパイラ技術の進歩とアプリケーションの多様化・分極化は、求められる計算機システムアーキテクチャを常に変化させてきた。

計算機システムアーキテクチャのなかでも、プロセッサ・アーキテクチャ(命令セットレベル・アーキテクチャとマイクロ命令レベル/レジスタ・トランスファーレベル・アーキテクチャ)は、計算機の本質とも言うべき処理性能を決定するものとして、非常に重要である。命令セットレベル・アーキテクチャとしては、種々の提案がなされて来ており、長い期間に渡って主要な位置を占めて来たIBM S/360～S/370～S/390命令セット・アーキテクチャや、独特の地位を占めて来たスタックマシンの命令セット・アーキテクチャ、最近では、コンパイラ技術の進歩に裏付けられた縮小命令セット・アーキテクチャ(RISC)などがある。

これらの命令セットレベル・アーキテクチャが主流になる/ならないは、工学的な評価だけで決まるわけではなく、多分に、企業の力関係や商業的理由によることが多い。しかし、純粋に、プロセッサ・アーキテクチャを工学的に評価するとするとその評価項目は、

- 性能
- コスト

である。即ち、性能/コストが最大となるものが最良である。しかし、性能もコストも、命令セットレベル・アーキテクチャの定義からでは計る事ができず、そのアーキテクチャを実現するプロセッサを設計・製造(あるいは、それをシミュレーション)して始めて知ることができる。そのため、純粋な命令セットレベル・アーキテクチャでの定量的比較はできないと言える。

このため、命令セットレベル・アーキテクチャとマイクロ命令レベル/レジスタ・トランスファーレベル・アーキテクチャを総合した、プロセッサ・アーキテクチャで評価/比較を行う事

になる。この時、性能/コストを決定するもう一つの要因は、使用するテクノロジである。テクノロジは、次のような側面で比較できる。

- 使用する論理素子のスピード
- 使用する論理素子の集積度
- 使用する論理素子の消費電力/発熱量
- 実装密度
- 冷却能力
- 電力供給方式

これらの項目は、互いに独立したものではなく相互に関連したものであり、「コスト」と言う尺度を定めれば、ある時代で定まるものである。即ち、ある時代を決めれば、それによってプロセッサ・アーキテクチャの性能とコストを確定することができ、どのプロセッサ・アーキテクチャが最良かが決められる。逆に、時代とともに、テクノロジは進歩するので、一つのプロセッサ・アーキテクチャが常に最良とは限らない。この点から、テクノロジの進歩がプロセッサ・アーキテクチャを発展させる原動力となっていると言える。

このような状況で、筆者は、約 10 年間に渡って、命令/操作レベルで並列処理機能を持つプロセッサを中心とした計算機システムを 2 種類開発する機会を得た。一つは、約 10 年前に開発した QA-2 であり、他方は、最近発表した富士通 VPP500 ベクトルパラレル・スーパーコンピュータの要素プロセッサである。この 2 つのマシンの開発時期は 10 年間の隔たりがあり、その間のハードウェア素子・実装技術の進歩は著しい。特に現在は、CMOS-LSI の高集積度化が目を見張る速度で進んでおり、価格も大量生産化により、どんどん低下して行っている。10 年前は、一部のスーパーコンピュータは特別として、汎用超大型計算機でも命令パイプライン処理を行っている程度で、命令レベル並列処理などは、殆どの計算機では行われていなかった。命令レベル並列処理は、プロセッサ・アーキテクチャとそれを裏付けるハードウェア技術の観点からすると、当時のテクノロジ・レベルでは少々オーバースペック気味であったかも知れない。しかし、現在では、数万円のマイクロプロセッサでも、スーパースカラ方式と言う命令レベル並列処理を行っている。この 2 機種の開発において、命令/操作レベルの並列処理機能を洗練し、現在のテクノロジー・レベルに適合したプロセッサ・アーキテクチャの構成法について実証できたと確信している。

本論文の構成は、大きく 2 部からなり、第一部では、10 年前に開発した QA-2 について述べる。QA-2 は、算術論理演算装置 (ALU)-レジスタ・レベルでの並列処理機能を有するマイクロプログラム制御計算機であり、旧型機である QA-1 における各種の応用実験を通じて行った性能評価に基づいて、そのハードウェア構成方式を改良したシステムである。QA-2 は、そのマイクロプログラムを書き換えることによって、図形処理や信号処理などのリアルタイム処理、および高級言語処理に対する効率の良いシステムを提供できる。QA-2 では、256 ビット長の水平型マイクロ命令の相異なるフィールドによって、4 個の可変長 ALU 演算、4 個の主記憶アドレスへのアクセス、1 個の高機能順序制御を同時に指定できる低レベル並列処理方式が採用されており、マイクロプログラム制御方式の柔軟性を生かして、多様な応用に対処できるようになっている。また、処理速度や適応能力のほかにマイクロプログラムの生産性を上げることをも考慮して、マイクロ・アーキテクチャを一様に構成している。

第 2 章では、QA-2 システムの設計思想を明らかにし、第 3 章では、その最大の特徴である低レベル並列処理機構の実現方式について述べる。また、実際に QA-2 を応用した結果を示し、低レベル並列処理方式の有効性について評価を加える。第 4 章では、主記憶部の構成について述べる。

また、QA-2 では、低レベル並列処理機構の高速性を生かし、しかもユニバーサル・ホスト計算機としての広範な問題適応能力を得るために、ハードウェア機構として様々な工夫を施した高機能順序制御方式を採用した。第 5 章では、この順序制御部 (SCU) のハードウェア構成について詳述し、その方式について簡単に評価を加える。

QA-2 システムでは、システム管理のために専用プロセッサ SVP も開発した。SVP は、(a) システム記述専用高級言語の処理に適した高速機械命令の実現、(b) マルチ・プロセスの効率の良い実行環境の実現、(c) ソフトウェアのモジュール化の支援、などを達成できるよう構成されている。このような高機能機械命令 (マクロ命令) の高速実行を実現するために、SVP では、(a) 簡潔な水平型マイクロ命令を用いた制御方式によって得られる柔軟性と高速性の利用、(b) システム管理に重要な役割を果たす主記憶管理機構・アクセス管理機構の高速化手法の導入、などを図っている。第 6 章では SVP のマイクロ/マクロ・アーキテクチャについて述べた後、OS の一部であるファイル管理プログラムを例にとって本システムで採用した方式の妥当性についても言及する。

QA-2 の開発において採用した各種の機能について、開発当時のテクノロジ・レベルでの評価は、それぞれの章で行っているが、それを、現在のテクノロジ・レベルで再評価を行うのは、それぞれの機能が普遍的な有効性を持っているかを知るうえで重要である。第 7 章では、この観点で QA-2 アーキテクチャの今日的評価を行う。

第二部では、最近、富士通株式会社から発表された、VPP500 ベクトルパラレル・スーパーコンピュータの要素プロセッサに採用された、スカラプロセッサ・アーキテクチャについて述べる。このプロセッサ・アーキテクチャは、QA-2 で試みた命令レベル並列処理と、富士通で開発してきたパイプライン処理のノウハウを融合させたものである。

第 8 章では、パイプライン処理の実際と、それを評価するためのプロセッサ性能の分析方法について述べ、VPP500 スカラ・アーキテクチャの開発にあたってのバックグラウンドを明らかにする。

第9章では、具体的にVPP500の命令セットレベル・アーキテクチャとレジスタ・トランスファーレベル・アーキテクチャを詳述し、簡単な評価を行う。

本研究で得られた主な成果は以下の通りである。

1. 命令/操作レベル並列処理方式の確立と有効性検証

   命令/操作レベル並列処理としては、現在、長命令語方式とスーパースカラ方式が一般的であるが、本研究では、長命令語方式の先駆的なシステムを開発し、その構成法を確立した。また、実際にハードウェアを作成して、具体的な実現法を明らかにした上で、その有効性を検証している。

2. 長命令語方式の並列性を低下させない順序制御方式の確立

   長命令語方式では判断が必要な条件の数で実行ステップ数が決まってしまい、並列に実行できる操作が存在しても性能向上が図れないと言う問題点が有った。この問題を克服する高機能順序制御方式を提案し、これにより順序制御で規定される実行ステップ数を削減できることを、実際のハードウェアに実装して明らかにしている。

3. 命令実行とオーバラップ実行するメモリ・アクセス方式の確立

   長命令語方式のように命令実行の高速化を行っても、主記憶アクセスは使用する記憶素子速度で制約され、プロセッサ全体の性能向上を阻害する。この問題に対する対策として、主記憶アクセスは、アクセスの起動以降の処理を突き放し処理で行うようなマイクロプログラム実行モデルを提案し、メモリ・アクセスとそれ以降のマイクロプログラム(命令)の実行をオーバラップさせて実効的に処理性能を向上する方式を確立した。

4. システム制御の負荷分散方式の確立

   アプリケーション・プログラムの実行とシステム制御を別プロセッサに負荷分散し、それぞれの実行に最適な命令セットレベル・アーキテクチャを選択したプロセッサをレジスタレベルで結合するシステム構成方法を確立した。

5. メモリ・プリフェッチ命令によるメモリ・アクセス性能向上方式の確立

   プロセッサの性能を決定づける要因に付いて分析し、命令/操作レベルの並列処理だけでは、性能向上に限界があることを明らかにした上で、性能向上を阻害する最大要因である、メモリ・アクセス待ちを削減する対策として、メモリ・プリフェッチ命令の導入を提案し、具体的な実現方法を明らかにしている。ここで提案している実現方法は、命令の実行モデルとして、命令の実行処理を同期部分と非同期部分に分けることによって、命令パイプラインの各種の待ち時間を削減し、かつ、制御ハードウェアは複雑にしないものであり、独創的である。

Part I

QA−2

Chapter 2

ＱＡ－２の設計思想

2.1　まえがき

LSI 技術の発展に伴い、計算機の利用形態は、従来の TSS による共有利用形態からホスト計算機をネットワークで結んだローカル計算機網による個人的利用形態に転換しつつある。この場合のローカル・ホスト計算機には、高級言語プログラムの高速実行および、図形、画像や信号のリアルタイム処理などが要求される。しかし、現在[1]市販されているマイクロプロセッサを基本としたワーク・ステーションでは、これらの応用に対して充分な性能を得ることができない。

ユニバーサル・ホスト計算機 [20] は、固定的なマシン命令セットを念頭に入れて設計された計算機ではなく、各種応用や言語に対して最適な中間言語を設定して、それをマイクロプログラムで解釈実行する計算機である。ユニバーサル・ホスト計算機として、Burroughs 社の B-1700/1800、Nanodata 社の QM-1、SCC 社の MLP-900、Stanford 大学の EMMY、電子技術総合研究所の PULCE などがある [20]。また、LISP マシンを始めとする高機能ワーク・ステーションには、この方式を基本にして設計されているものが多い [71]。しかし、マイクロプログラム制御による逐次実行のみでは大量データのリアルタイム処理に対処することが困難であるので、並列処理方式の導入が必要とされる。そこで我々は、マイクロプログラム制御方式との親和性が良い低レベル並列処理方式 (算術論理演算装置-レジスタ・レベルでの柔軟な並列処理方式) を考えて、これらを有機的に組み合わせることによって、ユーザの広範な要求を満たし得るユニバーサル・ホスト計算機 QA-2 を開発した。

QA-2 は、1976 年に完成した旧型機 QA-1[17] 上での種々の応用実験を通じての性能評価 [22][25][52] に基づいて設計されており、QA -1 を基に種々のアーキテクチャ上の工夫を施した計算機となっている [59]。 本章では、QA-2 の設計思想を明らかにし、特にその大きな特徴である低レベル並列処理機構の実現方式とその効果について述べる。

[1] 1980 年代のことである。

2.2 ユニバーサル・ホスト計算機と低レベル並列処理方式

QA-2 の適用分野は、図形処理や信号処理などのリアルタイム処理、および高級言語マシンなどの各種 (仮想) 計算機のエミュレーションであり、QA-2 は研究室の研究環境の中心となるホスト計算機として位置付けられている。ユニバーサル・ホスト計算機 [20] は、ソフトウェアによる処理をファームウェアで置き換えることによって、高速化を達成することを基本原理としている。従ってユニバーサル・ホスト計算機では、多様な応用に適応できるように、柔軟なマイクロプログラム制御方式が採用されている [20]。しかし、従来のユニバーサル・ホスト計算機の実行過程は逐次処理によっており、高級言語処理などの逐次処理向き応用の高速化を達成できる反面、図形・画像・信号処理などの処理対象に明示的な並列性を含む応用に対しては、十分な性能を提供できないものであった。

低レベル並列処理は並列演算方式の一種であり、図 2.1に示すように、比較的長い語長の命令の相異なるフィールドで多数の算術論理演算装置 (ALU)、メモリ・アクセス機構やバスなどをきめ細かく同時に制御する方式である。この方式は、演算機能レベルが低いので、マイクロプログラム制御方式との親和性が良い。我々は、QA-1 における経験から、処理対象に明示的な並列性のある応用 [22] に対してはもちろんのこと、高級言語処理やリスト処理などの明示的に並列性を有しない分野 [25][52] においても、柔軟なマイクロプログラム制御方式による低レベル並列処理方式が有効に適用できることを示してきた。また、処理性能の向上率、ハードウェア規模やマイクロプログラミングの容易さなどの観点からの考察により、並列演算可能な ALU の個数は 4 が適当であると判断した。QA-2 の設計思想は、QA-1 のそれを継承して、「マイクロプログラム制御方式と低レベル並列処理方式との有機的結合」を基本にしている。

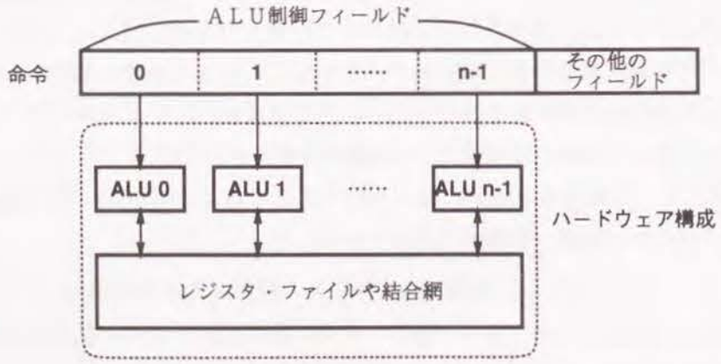


図 2.1: 低レベル並列処理方式

2.3 ユニバーサル・ホスト計算機のハードウェア構成方式

大規模マイクロプログラムの作成において、ユーザはプログラムの論理的構造 (演算式や制御構造によって記述するアルゴリズムやデータ構造) をマイクロプログラムが実行される物理的構造 (物理的なレジスタや主記憶空間におけるデータ表現、マイクロ命令の順序制御を行うハードウェア構成) に変換しなければならない。あるいは、高水準マイクロプログラム記述言語を用いる場合には、コンパイラが効率の良い目的マイクロプログラムに変換しなければならない。多様な応用に対して、この論理的構造と物理的構造のセマンティック・ギャップを出来るかぎり小さくできるハードウェア構成方式が、ユニバーサル・ホスト計算機として必要である。ユーザに対するマイクロプログラミングの負担が増大したり、高水準マイクロプログラム記述言語のコンパイラが普及しないのは、その計算機ハードウェアの設計時にマイクロプログラムの生産性についての配慮がなされていないことに起因している場合が多い [23] [13]。我々は、QA-1 の開発や応用によって得られた経験から、ユニバーサル・ホスト計算機の評価の基準として、処理速度や適応能力と同様に、ファームウェアの生産性をハードウェア構成方式の設計時にも考慮することが重要であると判断した。従って、QA-2 のマイクロ・アーキテクチャの実現においては、その並列演算機構や順序制御機構が一様な構造を保つように配慮した。

2.4 ALU の構成方式と低レベル並列処理

計算機アーキテクチャは、その命令起動方式の違いに応じて、コントロール駆動方式、データ駆動方式、要求駆動方式に大別される。このうちコントロール駆動方式では、制御装置は唯一であり、ALU の構成方式によって、単一命令・単一データ流 (SISD) 方式、スカラ並列演算方式、単一命令・複数データ流 (SIMD) 方式、演算パイプライン方式、およびこれらを複合化した複数命令・複数データ流 (MIMD) 方式の一形態であるマルチプロセッサ方式に分類される。このうちスカラ並列演算方式には、CDC6600 に代表される方式と、QA-2 に代表される低レベル並列処理方式がある [60]。CDC6600 で採用された方式では、命令パイプラインによって次々に生成される命令が複数の ALU で実行され、演算結果は命令パイプラインに入った順序とは異なった順序で出力され得る。CDC6600 では、機能の相異なる 10 個の ALU がレジスタを共有しながら、スコアボードの制御の下で並列演算を実行する。スコアボードは ALU やレジスタの状態を保持するものであり、命令の実行に際して必要な ALU の割り付けや、必要なオペランド・データが既に他の ALU やメモリから供給されているかどうかなどの検査を行い、命令が実行可能である場合には所定の ALU の実行が開始される。従って、この方式は、連続する命令の実行に際してデータ駆動方式で制御を行っていると言える。[60]

低レベル並列処理方式は、図 2.1に示したように、比較的長い命令を多数のフィールドに分割し、各々のフィールドで ALU などの機能装置を独立に制御する方式である。データ駆動方式や CDC6600 が実行時に並列演算の可能性を判定するのに対して、この方式のマシンではそ

れをコンパイル時に行う。コンパイラは、ソース・プログラムから並列演算可能なものを抽出して、一つの命令に合成する。その制御装置は、データ駆動方式などと比較して単純化できる。また、ALU などの機能装置間の通信は命令内のフィールドで直接制御され、そのオーバヘッドは少ない。この方式では、ALU 個数に近い並列度が得られる時に高速処理を達成できる。しかし、並列度が低い時には ALU 制御フィールドに遊びができ、命令のビット使用効率が低下する。また、複数 ALU で多数のステータスが生成されるので、順序制御部を柔軟な構成にしておく必要がある。低レベル並列処理方式を採用したマシンとして、表2.1のようなものがある。これらの大半は、単一の応用目的を意識して、非均質の ALU 構成 (機能の異なる多数の ALU を実装) 方式を採用している。しかし QA-2 では、ユニバーサル・ホスト計算機として、より広範な応用に適応できるように、均質な (同一機能の複数 ALU を実装)ALU 構成方式を採用している。

2.5 QA-2 システムの構成方式

ファームウェア工学 [13] の観点に立脚して設計された QA-2 システムの構成方式の特徴は次のようにまとめられる。

処理機能を分散化したシステム構成方式

図 2.2に示したように、QA-2 はレジスタ・ALU 部 (RALU) と順序制御部 (SCU) から成る CPU と、主記憶管理プロセッサ (MMP)、システム管理プロセッサ (SVP) の 3 つの機能部分を独立に構成し、これらを相異なるクロックと水平型マイクロ命令 (1 ワード=256 ビット) の相異なるフィールドによって独立して制御する機能分散方式によって構成されている。

同一で高機能な 4 個の ALU による低レベル並列処理方式

QA-2 では、図 2.3に示すように、4 個の可変長 ALU が 1 ワード (256 ビット) の水平型マイクロ命令の相異なるフィールドで独立に制御され、それらが均一で大容量のレジスタ・ファイル (6K バイト) を共有して動作する。4 個の ALU は互いに独立な 4 つのオペランド群に対して並列演算を実行できるだけでなく、1 個の演算結果を他の ALU 演算の入力オペランドとして使用する ALU 連鎖演算も可能である。

マイクロプログラムの生産性の向上を指向した順序制御構造

大規模なユーザ・マイクロプログラミングを実現するためには、ユーザが記述する論理レベルのマイクロプログラム構造と物理的なハードウェア構造との対応関係を容易にとり得ることが要求される。特に、高度な制御構造は構造化マイクロプログラミングを実行する場合に必須であり、“IF-THEN-ELSE” 文や “CASE-OF” 文などの強力な条件判定分岐機能はハードウェ

名称	発表年	開発機関	文献	応用分野	命令ビット長	命令実行サイクル nsec	並列 ALU 演算数	ALU の種類と特徴
AMP	1973	Argonne National Laboratory	[7]	リアルタイム処理 (グラフィックス等)	74	430	2	・8 ビット及び 16 ビット整数演算器
QA-1	1976	京都大学	[17]	図形・画像・信号処理、高級言語処理	160	350	4	・16 ビット整数演算器 (乗算器付き)
AP-120B	1976	Floating Point Systems	[15]	アレイ演算 信号処理 (FFT 演算など)	64	167	3	・38 ビット浮動小数点加算器 乗算器 (パイプライン制御) ・16 ビット整数演算器 (逆 2 進変換器付き)
Vanguard 1100/60	1979	Univac	[38]	汎用	283	116	2	・38 ビット汎用演算器 (MC -10800 を使用)
MUNAP	1979	宇都宮大学	[5]	非数値処理 (記号処理など)	188	555	4	・16 ビット非数値処理用プロセッサユニット ・2 レベルマイクロプログラム制御
QA-2	1983	京都大学	[26]	図形・画像・信号処理、高級言語処理	256	600	4	・8/16/24/32 ビット整数演算器 (乗算器付き)
ELI-512	1983	Yale Univ.	[14]	科学技術計算	500 以上	-	20～30	・32 ビット整数演算器 ・64 ビット浮動小数点演算器 ・Trace Scheduling によるコード・コンパクション
ベクトルプロセッサ	1984	日立製作所	[1]	(ミニコン内蔵型) ベクトル演算	96	167	3	・32 ビット浮動小数点乗算器 (パイプライン制御) ・64 ビット浮動小数点加算器 (パイプライン制御) ・32 ビット整数演算器
CYBER-PLUS	1984	CDC	[8]	科学技術計算	240	20	8	・8/16/32 ビット 整数加算器 (2 台)、 整数乗算器 (1 台)、 論理演算器 (2 台) ・32/64 ビット 浮動小数点加算器、 乗算器、 除算 (平方根演算) 器 (パイプライン制御、各 1 台)
MC	1985	松下電器産業・無線研究所	[39]	グラフィックス	96	250	5	・浮動小数点加算器、乗算器 ・関数発生器 (三角関数、平方根、逆数など) ・浮動-固定小数点数変換器 ・24 ビット汎用演算器

表 2.1: 低レベル並列処理方式のマシン

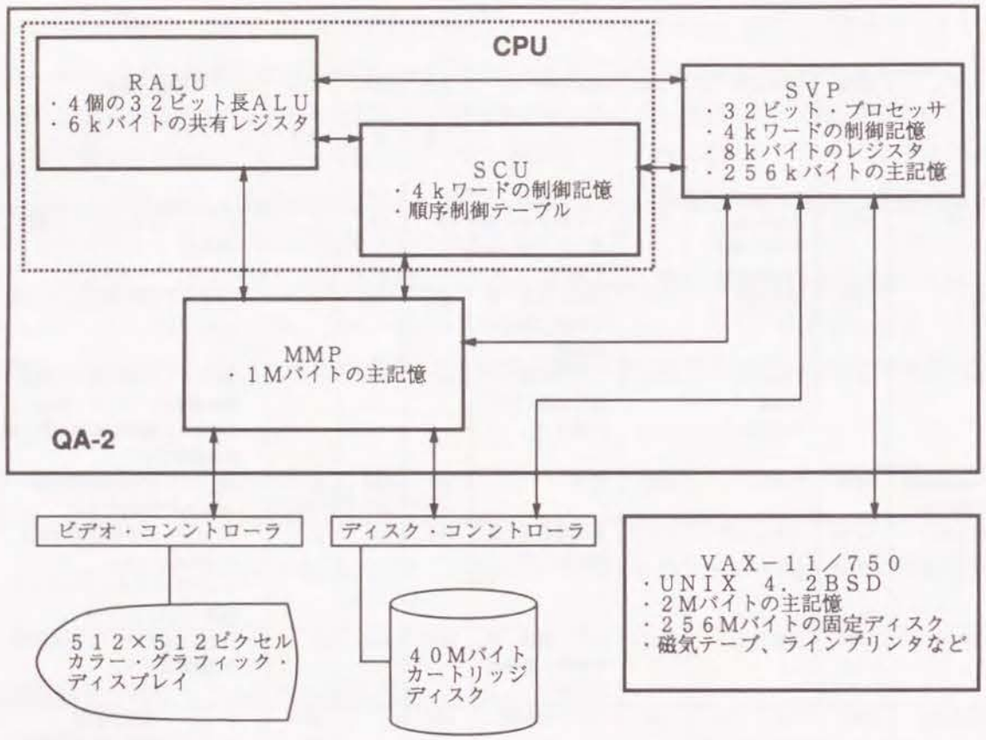


図 2.2: QA-2 のシステム構成

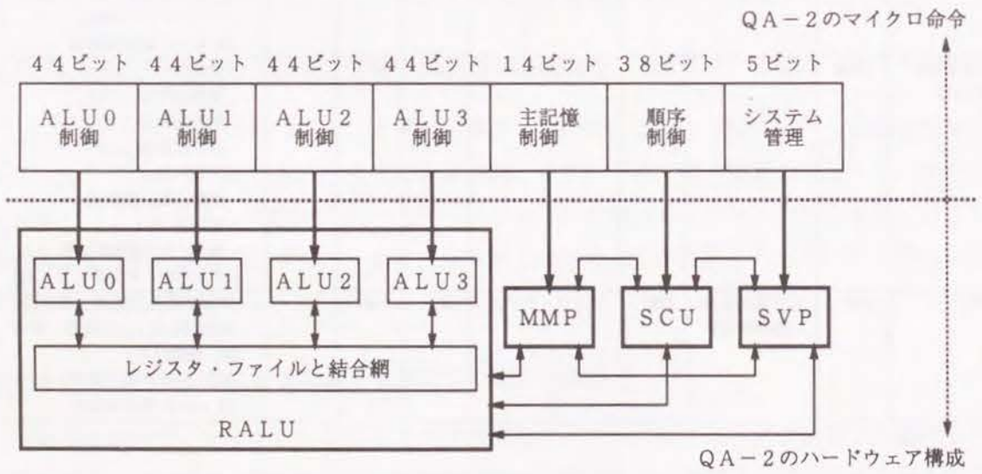


図 2.3: QA-2 の低レベル並列処理方式

アとして実装されていることが望ましい。QA-2 では、低レベル並列処理機能を十分に生かせるように、高度の順序制御構造が採用されている。

Chapter 3

レジスタ・演算部の構成

3.1 QA-2 の低レベル並列処理機構

図 3.1に示したように、QA-2 の CPU は RALU と SCU との 2 つの機能部分に分けられる。さらに、RALU は 1 個の均一構成のレジスタ・ファイル (RF) と 4 個の同一機能の ALU とから成る。

3.1.1 均質・大容量共有レジスタ・ファイルの構成方式

低レベル並列処理方式では、ALU がレジスタを共有して動作する。従って、共有レジスタの構成方式がプログラミングの負担とシステム性能に大きな影響を及ぼす。理想的には、共有レジスタは大容量で、しかも任意の ALU の左右入力にデータを供給でき、任意の ALU の出力結果を格納できることが望ましい。

QA-2 では、以下のような方法で均質・大容量の共有レジスタを実現している。QA-2 の RALU に装備されているレジスタ群としては、図 3.1に示すように、定数レジスタ・ファイル (CRF: 1K バイト)、汎用レジスタ・ファイル (GRF: 1K バイト)、間接レジスタ・ファイル (IRF: 1K バイト×4 バンク)、および特殊レジスタ (SR) がある。このうち GRF、CRF、IRF は 4 個の ALU のいずれからも一様な共有レジスタ空間としてバイト単位でアクセスすることができるので、ユーザによる複雑なレジスタ割り付けは不要である。また、これらのレジスタ・ファイルは ECL・RAM で実装されており、実際のハードウェア構成として、各 ALU の右入力、左入力ごとに各々同一コピーをもつレジスタ・ファイルが設けられている。従って、4 個の ALU 演算のソース・オペランドとして相異なるアドレスに存在する 8 個のデータを、同時に読み出すことが可能である。また、図 3.2に示すように、ALU 演算終了時にはこれら合計 8 枚のレジスタ・ファイルすべてにリング状の ALU 出力パスを通して順々に 4 個の演算結果が格納され、これらのレジスタ・ファイルには最終的に同一データが保持される。レジスタ・ファイルの各ポートは 4 バンク構成になっており、複数バイト・アクセスの際には、バイト整列がハードウェ

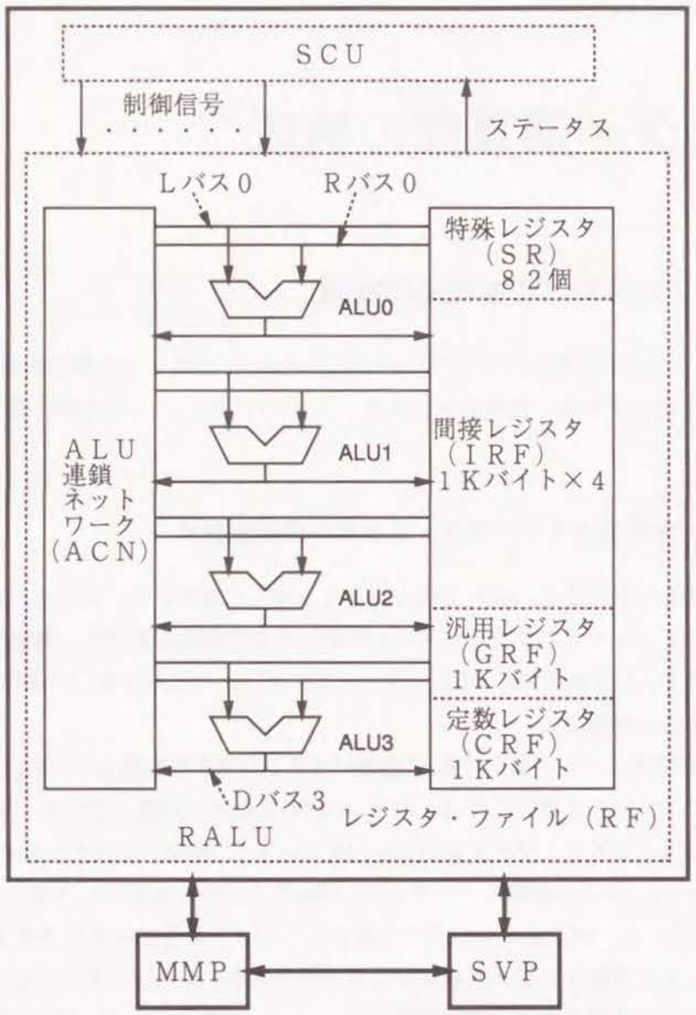


図 3.1: RALU のバス構成

アでなされる (図 3.2のバイト整列回路)。

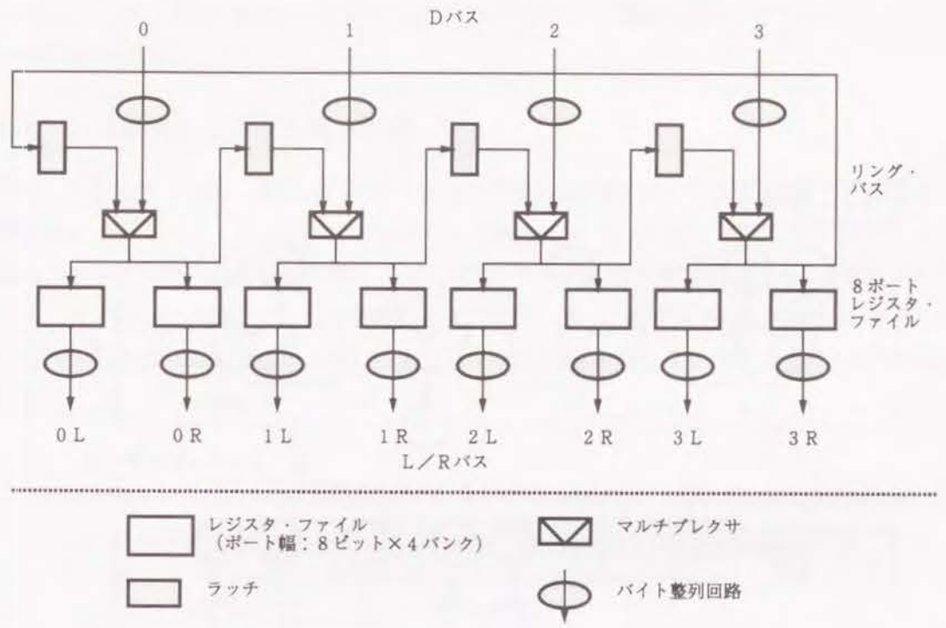


図 3.2: レジスタ・ファイルの構成方式

3.1.2 均質な ALU 構成による並列/連鎖演算機構

低レベル並列処理方式では、マイクロ命令の各フィールドが活用されて初めて、高速処理を達成できる。従って、ユーザが論理レベルで考えた並列演算構造をそのまま物理レベルの構造に対応付けることが可能でなければならない。さらに、性能を上げるためには、最適化手法による並列操作の 1 マイクロ命令への合成 (埋めこみ) を行うことが必要である。

QA-2 の各 ALU での演算長は、バイト単位に最大 4 バイトまで可変であり、レジスタへのアクセスも演算データ長に合わせて自動的に制御される。さらに QA-2 では、図 3.3に示したように、1 個の ALU による演算結果を別の ALU の入力データとして指定する ALU 連鎖演算機能も備えているので、ユーザが 4 個までの ALU 演算を並列/逐次の区別なく 1 マイクロ命令で記述できる。従って、1 マイクロ命令で実行できる ALU 演算機能レベルが向上し、データの一時退避というオーバヘッドも無くなっている。1 マイクロ命令で最大 4 組の演算の実行が保証されているので、ユーザ・マイクロプログラムの論理構造と物理的な ALU 構成との対応付けが容易であり、低レベル並列処理機能を活用した大規模ユーザ・マイクロプログラミングが可能である。

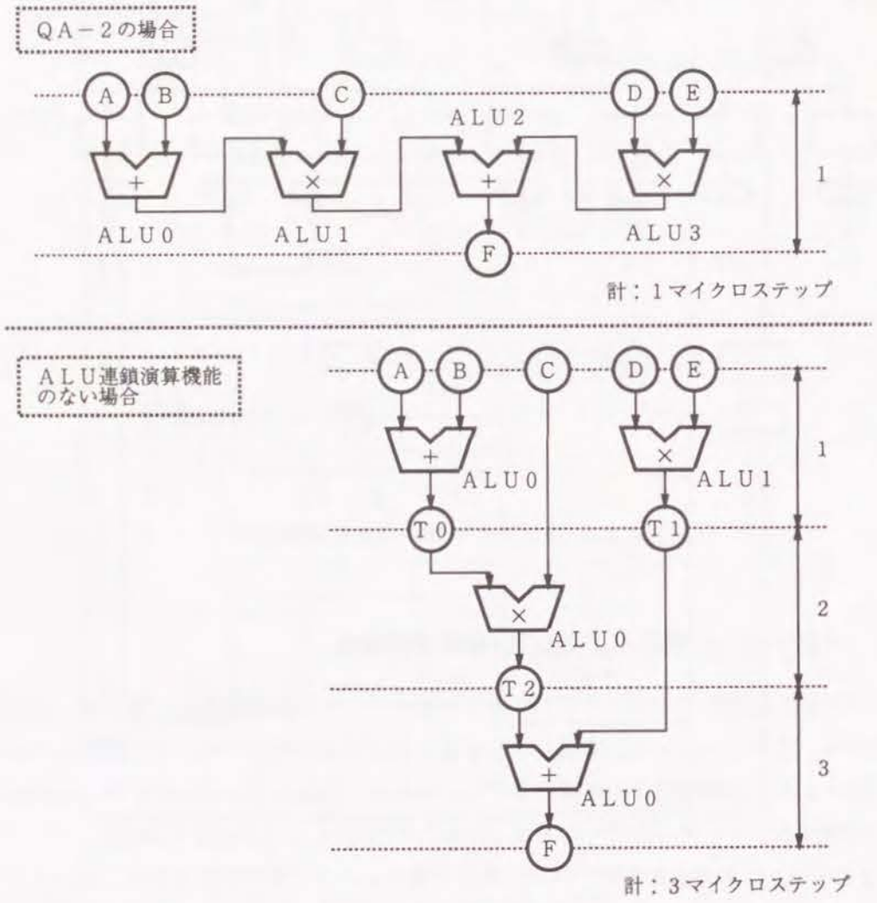


図 3.3: ALU 連鎖演算機能の効果

例えば図 3.3の例では、変数 A～F をすべて共有レジスタに割り付けて、ALU 連鎖演算機能を用いると、この演算: "$F = (A + B) \times C + D \times E$" を 1 マイクロ・ステップで行うことが可能となる。しかし ALU 連鎖演算機能を持たない場合には、同様の演算を行うのに 3 マイクロ・ステップ必要である。

3.1.3 バス構成とマイクロ命令形式

各 ALU 入力ラッチは、図 3.4に示すマイクロ命令の LSRC および RSRC フィールドによって指定されたレジスタ・ファイル (RF) 内のいずれかのレジスタと L バスおよび R バスによって結合される。ALU 演算の結果である ALU 出力データは D バスを経由して、DST フィールドによって指定されたレジスタに格納される。また、各 D バスは ALU 連鎖ネットワーク回路 (ACN) を経由して任意の他の ALU 入力ラッチとも結合することができる (ALU 連鎖演算機能)。

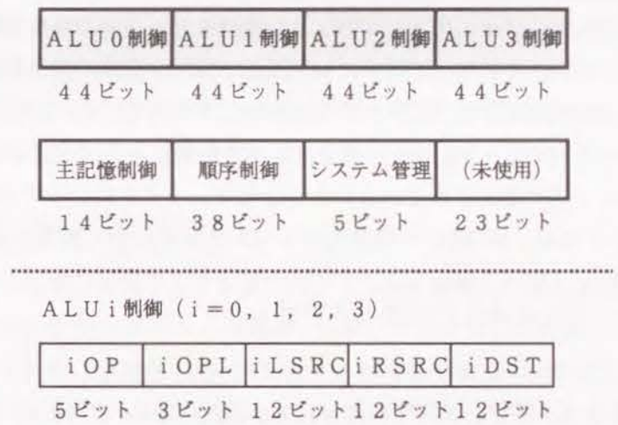


図 3.4: マイクロ命令形式

各 ALU オペランドのデータ長は OPL フィールドで指定され可変 (1、2、3 または 4 バイト) である。従って、ユーザは 1 マイクロ命令で 4 個の相異なる可変長データに対する並列 ALU 演算を指定できる。ALU への入力データはバイト単位でアクセス可能な RF から、右端 (最下位ビット) 調整されて読み出される。

3.1.4 ALU 演算機能とその制御方式

マイクロ命令の OP フィールドが ALU による演算を指定する。複数 ALU 演算を連鎖演算機能によって組み合わせることができるので、高機能 ALU 演算 (例えば、データ・フィールド操

作、ビット入れ換え操作あるいは浮動小数点数の正規化など) も 1 マイクロ命令で実行することが可能である。OP フィールドで指定できる特徴的な ALU 演算には次のようなものがある。

- 16 ビット・ECL バレルシフタ (MC10808) による高速並列シフト (最大 32 ビット) 演算。シフト桁数は、RSRC フィールドで指定する。
- 8×8 ビット・乗算器 (Am25S558) による最大 32 ビット長整数の乗算。符号付きと絶対値表示の 2 種類の整数乗算が可能である。
- 浮動小数点演算を支援するためのプライオリティ・エンコード (データの最上位ビットから連続する "0" や "1" のビット列長の計数) 演算。
- マスク機能付きのバイトあるいはビット演算。鏡像 (逆 2 進) 演算。
- 専用レジスタの内容による間接演算指定。

QA-2 の各 ALU 機能モジュールは 32 ビット長演算を基本に設計されている。また ALU 部には、例えば浮動小数点プロセッサ・チップなどの機能モジュールの付加を簡単に行うことができる構成になっている。そして、可変長データ演算や実行時間長の異なる演算に対処するために、機能モジュールの各部分は、演算タイミングなども含めて、ナノ記憶 (256 ワード) から読み出された 1 ワード (64 ビット) のナノ命令によって制御する方式を採用している。ナノ記憶は ECL・RAM チップで構成されている。ナノ命令は、マイクロ命令の OP と OPL フィールドに従ってフェッチされ、各 ALU の機能モジュールをきめ細かく制御する。

3.1.5 レジスタ・ファイルへのアクセス方式

レジスタ・ファイル (RF) に対するアクセス方式は (図 3.5参照)、マイクロ命令の LSRC、RSRC および DST フィールドによって指定される。GRF、CRF と SR への直接アクセス (図 3.5の 1) は、各フィールドの下位 10 ビット (X9,~X0) によってアドレス指定が行われる。CRF にはマイクロ・アセンブラによって必要と判定された定数が、マイクロプログラムのロード時に SVP によって格納される。9 ビット以上の定数は LSRC または RSRC フィールドによって指定されたアドレスの CRF から読み出されるが、8 ビット以下の定数は直接マイクロ命令のフィールドを利用して生成され (図 3.5の 2)、CRF には格納されない。

IRF へのアクセス (図 3.5の 3) は 12 個ある IRF 用アドレス・レジスタ (IRFAR) のうち 1 個を LSRC、RSRC あるいは DST フィールドの下位 4 ビット (Z3,~Z0) によって選択することによって行う。また、(Z5,Z4) によって IRF アクセス時の IRFAR の自動増減機能を制御する。IRFAR はアクセスされたバイト長だけ自動的に増減されるので、ユーザは IRF を論理的なスタックあるいはキュー構造のレジスタ空間とみなして利用することができる。IRF は 4 組装備されており、そのうちの 1 組を (Z7,Z6) で選択する (ただし、Z8=0 の場合)。なお、4 個の IRF

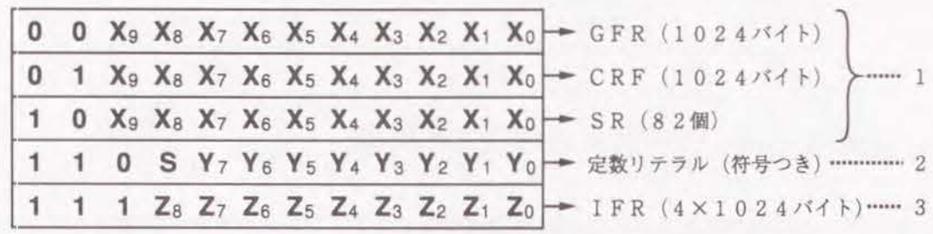


図 3.5: マイクロ命令フィールドによるレジスタ・ファイルの指定

を連続した 4K バイトの大容量レジスタ空間としたアクセス・モードにすることも可能である (Z8=1 の場合)。

SR は QA-2 内部のハードウェア・ファシリティを直接制御するために設けられており、各々が独立した MSI チップによって構成されている。SR には、例えば主記憶管理用にメモリ・アドレス・レジスタやメモリ・データ・レジスタ、順序制御用にカウンタ、さらに RF アクセス制御用に前述した IRFAR などが装備されている。また、RF アクセスに対する間接アドレス指定も可能となっている。この場合には、RF 内の 1 ワードがまず読み出され、その内容 (図 3.5と同一形式) に基づいて、RF へのアクセスが再度実行される。

3.1.6 レジスタ・ALU 部の制御方式

RALU では、異なる実行時間を有する演算のタイミング制御、LSRC/RSRC/DST フィールドによる直接アクセスや IRF アクセスの制御、間接アドレス指定の制御、ALU 連鎖演算の制御などが必要である。RALU の各 ALU に対して、LSRC、RSRC、EXEC、DST 制御オートマトンを配置し、これら合計 16 個の制御オートマトンが相互に同期をとりながら状態遷移し、制御信号を生成するようになっている。

例えば、ALU0 の OP フィールドで乗算、LSRC で IRF アクセス、RSRC で ALU1 の演算結果、DST で間接アドレスをそれぞれ指定した場合における、ALU やレジスタの制御信号について説明する。この例では、マイクロ命令がフェッチされると、ALU0 の LSRC および RSRC 制御オートマトンがまず起動される。LSRC 制御オートマトンは、IRFAR の内容を読み出し、IRF にアクセスし、指定されたデータをラッチする。RSRC 制御オートマトンは、ALU1 の EXEC 制御オートマトンから演算終了を通知されるまで待ち状態となり、ALU1 の演算が終了するとその結果をラッチに取り込む。ALU0 の演算に必要な両 SRC データが揃うと、ALU0 の EXEC、DST 制御オートマトンが起動される。DST 制御オートマトンは、間接アドレスの計算を行って、全 ALU の EXEC 制御オートマトンからの終了通知を待つ。EXEC の終了信号が揃うと、全 ALU の DST 制御オートマトンは一斉に、RF に対する書き込み (図 3.2参照) ス

テージに入る。ALU0 の EXEC 制御オートマトンは、ナノ命令の情報を使って乗算の演算時間長だけ待ち、全 ALU に終了通知を発する。

3.2 QA-2 の低レベル並列処理方式の評価

QA-2 の低レベル並列処理方式の有効性を確認するために、QA-2 を実際の問題に応用し、動的な性能評価を行った。例題としては、処理対象に明示的な並列性のある応用として、1) スキャンライン・アルゴリズムによる 3 次元グラフィックス、また明示的に並列性を有しない応用として、2) 逐次型 Prolog のインタプリタ、3) Lisp のインタプリタ、を選んだ。いずれの応用においても、QA-2 のマイクロ・アーキテクチャについて様々な観点から定量的な評価を加えたが、本章では QA-2 の低レベル並列処理機能の効果の概要を述べる [62]。 各応用における QA-2 の代表的な特性を抽出するために、1) では 30 個の立方体をランダムに表示する場合の可視部分の決定ルーチン、2) では 30 個の要素から成るリストを反転するプログラム、3) では「tarai-4」のベンチマーク・プログラムを、それぞれ具体的な問題として選んだ。 まず、表 3.1 に示すように、1 マイクロ命令で使用されている平均 ALU 個数は、いずれの問題においても約 3 個であり、問題における明示的な並列性の有無にかかわらず、QA-2 の低レベル並列処理機能を充分に活用できることが分かった。

応用問題	(個/4 個)
(1) 3 次元グラフィクス	3.04
(2) Prolog インタプリタ	2.93
(3) Lisp インタプリタ	2.93

表 3.1: 1 マイクロ命令における平均使用 ALU 個数

また、QA-2 から ALU 連鎖演算機能を削除すると、各問題において 20〜50%のオーバヘッドが生じる (表 3.2参照)。特に、明示的な並列性を持たない Prolog マシンや Lisp マシンのエミュレーションにおいては、この機能の効果が大きいことが分かる。

さらに表 3.2に示すように、QA-2 において 4 個の ALU を並列動作させる効果は、1 個しか ALU を持たない場合の 2.5〜3 倍もあり、逆に ALU を 8 個 (2 倍) にしても、数%からせいぜい 20%の性能向上しか得られない。ALU を 8 個にした場合、ハードウェア規模は確実に 2 倍以上になることを考慮すると、並列演算可能な ALU 個数としては 4 が妥当であり、広範な問題適応性を持つと判断できる。

応用問題	ALU 個数				連鎖演算
	1	2	4(現状)	8	機能なし
(1) 3 次元グラフィクス	3.04	1.66	1	0.87	1.20
(2) Prolog インタプリタ	2.94	1.65	1	0.79	1.38
(3) Lisp インタプリタ	2.51	1.65	1	0.97	1.49

表 3.2: ALU 構成を変えた場合の実行マイクロ命令ステップ比

3.3 むすび

QA-2 は、旧型機 QA-1 による応用で得られた種々の定性的・定量的評価データを基にして新しく設計されたものであり、マイクロ・アーキテクチャはユニバーサル・ホスト計算機として高性能かつ柔軟性に富んだ斬新なものとなっている。特に、その低レベル並列処理機能の有効性は、各種の応用によって確認されている。

しかし、図 3.6の全体写真で示したように、実装規模は予想外に大規模化し (QA-2 全体での IC チップ数は約 22,000 個)、最小マイクロ命令実行サイクルは約 600 ナノ秒 (設計時の性能予測では、200 ナノ秒) となっている。QA-2 のハードェアは、メモリ (MOS チップ) を除いて ECL とショットキ TTL チップで構成されている。乗算器やシフタなどの ALU 関連あるいは ECL メモリで構成されている汎用レジスタ・ファイルなどでは LSI が使われているが、ビットスライス・マイクロプロセッサは使用していない。図 3.7に筐体の実装状況、図 3.8にカード上の実装状況を示す。また、システムの設計開始からハードウェア本体の開発の終了まで 4 年半を要した。

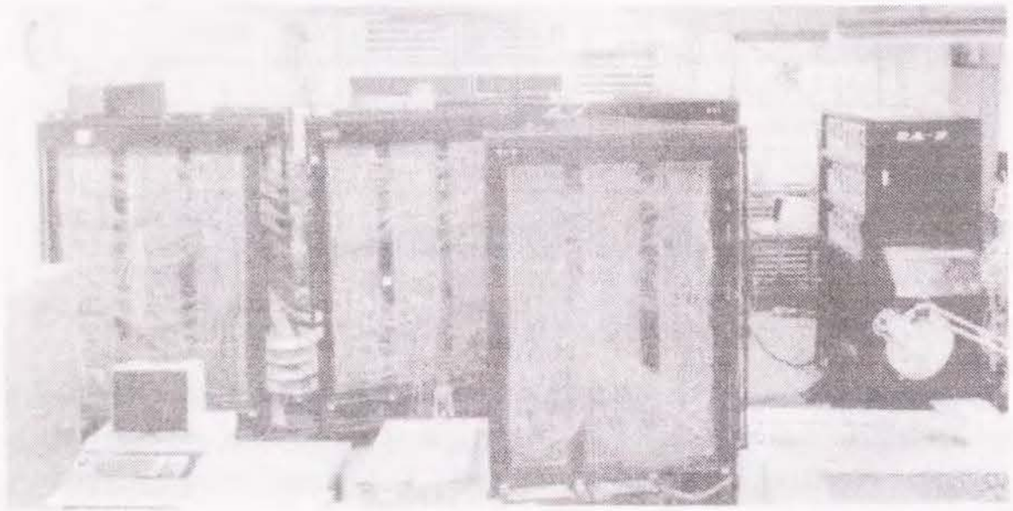


図 3.6: QA-2 の全体写真

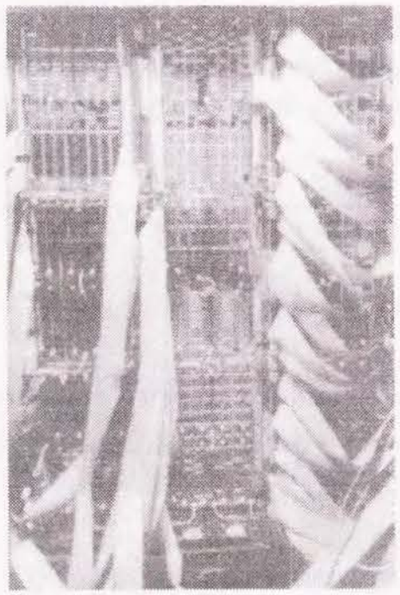


図 3.7: 1 筐体の実装状況写真

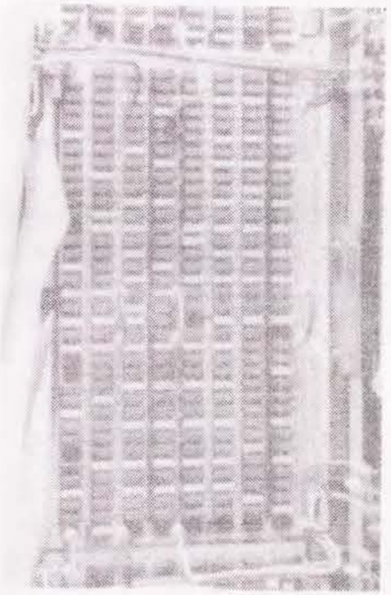


図 3.8: カード上実装状況写真

Chapter 4

順序制御部の構成

4.1 まえがき

我々は、低レベル並列処理機能をマイクロプログラム方式で制御するというシステム構成法のもとに、高速性と柔軟性を兼備した実験用計算機:QA-2 を開発した。QA-2 は、図形・画像・信号データのリアルタイム処理や高級言語処理を主要な応用対象としたユニバーサル・ホスト計算機である [42]。

QA-2 の設計思想は、「低レベル並列処理方式とマイクロプログラム制御方式の有機的結合」であり、QA-2 には処理速度、システムの柔軟性、およびマイクロプログラムの生産性を飛躍的に高め得るように、高機能かつ一様な構造の並列演算機構、メモリ・アクセス機構、順序制御機構が採用されている。QA-2 では、図 4.1に示すように、レジスタ・ALU 部 (RALU) と順序制御部 (SCU) から成る CPU と、主記憶管理プロセッサ (MMP)、システム管理プロセッサ (SVP) の 3 つの機能部分を独立に構成している。CPU と MMP は、図 4.2に示した水平型マイクロ命令 (1 ワード=256 ビット) の相異なるフィールドによって制御される。SVP は、独立した 32 ビット・プロセッサである [43]。

同一構造で高機能な複数 ALU による低レベル並列処理機能とは、4 個の可変長 ALU が水平型マイクロ命令 (1 ワード=256 ビット) の相異なるフィールドで独立に制御され、それらが均一構造を持ちかつ大容量 (6K バイト) のレジスタ・ファイルを共有しながら動作する方式のことである。4 個の ALU は互いに独立な 4 つのオペランド群に対して並列演算を実行できるだけでなく、1 個の演算結果を他の ALU 演算の入力オペランドとして使用する ALU 連鎖演算を行うことも可能である。

本章ではユニバーサル・ホスト計算機として実現した QA-2 の順序制御方式に焦点を絞り、詳述する。

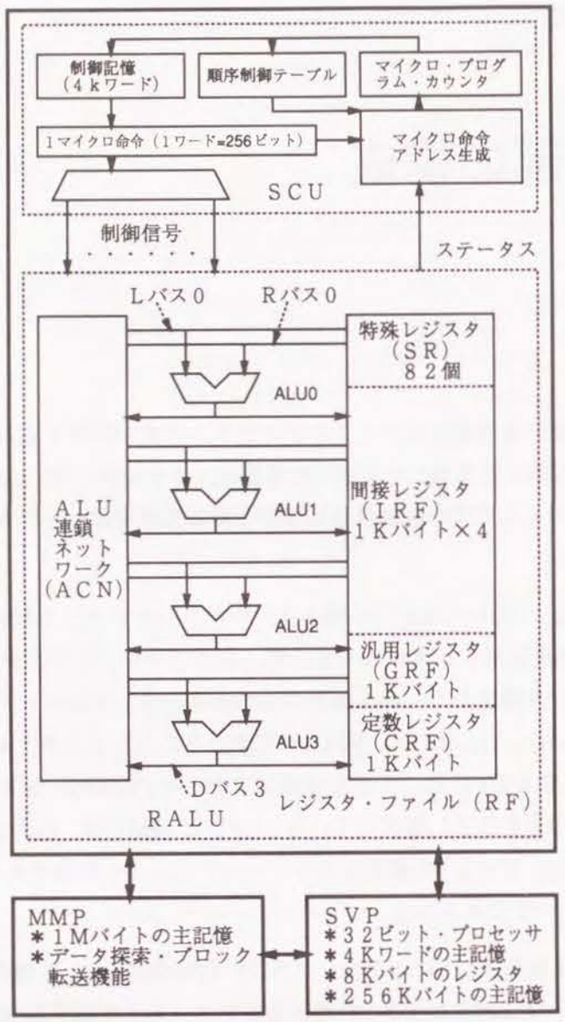


図 4.1: QA-2 のシステム構成

１マイクロ命令（２５６ビット）

ＡＬＵ０制御	ＡＬＵ１制御	ＡＬＵ２制御	ＡＬＵ３制御
４４ビット	４４ビット	４４ビット	４４ビット

主記憶制御	順序制御	システム管理	（未使用）
１４ビット	３８ビット	５ビット	２３ビット

順序制御：

ＳＣＳ	ＳＴＹ	ＳＭＣ	ＡＤＧ
１ビット	２ビット	８ビット	２７ビット

無条件分岐：

ＩＤＪ	ＡＤＳ	ＤＩＳＰ	ＡＳＳ	ＡＳＤ
１ビット	４ビット	１５ビット	４ビット	３ビット

２方向分岐：

ＳＰＣ	ＩＦＪＡ	ＩＦＪＣ	（未使用）
８ビット	１０ビット	８ビット	１ビット

多方向分岐：

ＳＰＣ	ＣＪＢＡ	（未使用）
８ビット	１２ビット	７ビット

図 4.2: マイクロ命令形式

4.2 ユニバーサル・ホスト計算機の順序制御方式

ユニバーサル・ホスト計算機におけるマイクロプログラムの順序制御機構に対して一般的に要求される機能は、高速性と柔軟性 (問題適応性) につきる。特に QA-2 では、低レベル並列処理方式を採用していることにより、これらの要求は次のような特殊な状況として具体化してきている。

- 大規模なユーザ・マイクロプログラミングにおける生産性を向上させるために採用した ALU 演算や順序制御機構を、ユーザが構造化マイクロプログラミングにおいて活用できなければならない。
- 問題適応性を高めるために、できるだけ自由度のある (ハードウェア構成による制限が少ない) 分岐形式の選択、条件ステータスの抽出、および条件判定用論理関数の指定などが可能でなければならない。
- 並列・連鎖 ALU 演算により、1 マイクロ命令の実行で多数のステータスが生成されるので、これらを十分に活用できる条件分岐機能が実現されなければならない。
- レジスタ・ALU 部と順序制御部の機能バランスを考えて、4 個の ALU 演算フィールドができるだけ遊ばないようなマイクロプログラミングが可能でなければならない。

4.2.1 低レベル並列処理方式と順序制御機能

マイクロ命令内の並列処理度が多い QA-2 の場合には、各種の論理変数の組み合わせによる複雑な条件テストの出現頻度が増大するので、組織的な条件テスト部を構成しなければならない。2 方向分岐 (IF-THEN-ELSE 分岐) 方式としては、その制御に必要なマイクロ命令のフィールドが冗長になるのを避けて、ELSE 節を明示しない形式や ELSE 節がリテラル・フィールド (分岐先アドレス) を必要としない形式などが採用されることが多い。しかし、THEN 節と ELSE 節の分岐形式を同一構造にすることは、マイクロプログラムの生産性を向上させる効果もあり、QA-2 ではこの形式を採用している。

例えば、図 4.3に示したような 2 方向分岐とサブルーチンコールとが混在する順序制御においては、これを QA-2 では 1 マイクロ・ステップで実現できるのに対して、"IF (X) THEN GOTO A ELSE NEXT ;" 形式という単純な 2 方向分岐機能しか持っていない計算機では、サブルーチン・コールが起動されるまでに 3 マイクロ・ステップを要する。

また、低レベル並列処理計算機の場合には、ステータス・レジスタの内容に応じた多方向分岐 (CASE-OF 分岐) が効率の良いマイクロプログラムの実行に本質的な役割を果たす。図 4.4 に示したように、QA-2 の場合には 1 マイクロ・ステップで終了するようなマイクロプログラムでも、単純な 2 方向分岐機能しか持たない計算機では 4 マイクロ・ステップが必要である。多方向分岐機能としては、ステータスの値をそのままマイクロ命令のリテラル・フィールドの

単純な順序制御方式の場合：

```
IF (A0) THEN GOTO L ELSE NEXT;
IF (A1) THEN GOTO L ELSE NEXT;
CALL S;
```

QA－2の場合：

```
IF f (A0,A1)=(A0|A1) THEN GOTO L
                     ELSE CALL S;
```

図 4.3: 複雑な順序制御の指定例

値と連結したり加算したりしてマイクロプログラム・カウンタの値とする方式や、ステータスの値を用いた ALU 演算により直接マイクロプログラム・カウンタの内容を変更する方式などがあるが、これらの方式では分岐形式や分岐先の指定に対する制限が強くならざるを得ない。QA-2 では、ステータス・レジスタ内の任意ビットを切り出してマイクロプログラム・カウンタの値の設定に使用できる機構を装備し、さらにディスパッチ・テーブルに格納する情報を豊富にして、各条件値に対する多様な分岐形式の指定を柔軟に行うことができるようになっている。

4.2.2 構造化マイクロプログラミング

ユニバーサル・ホスト計算機上で大規模マイクロプログラムを作成するためには、ソフトウェア工学で提唱された構造化手法を適用する必要がある [23] [13]。しかし、マイクロプログラミングの場合には、ハードウェアの直接的な高速制御機能を実現しなければならないので、構造化手法の適用がシステムの性能低下を招かないように注意する必要がある。従って、構造化マイクロプログラミングのためには、相応の順序制御機構をハードウェアとして用意しなければならない [60]。 また、マイクロプログラムはハードウェアをきめ細かく制御するものであり、通常のマシン命令プログラムと比較して、分岐命令の出現頻度が極めて高い。特に、水平形マイクロ命令形式で多数の機能装置を並列に制御する方式を採っている QA-2 の場合には、この傾向はより顕著になっている。高水準マイクロプログラム記述言語の利用は、マイクロプログラミングにおける生産性を高める一方式であるが、マイクロ・アーキテクチャが複雑で非均質な構造を持つ場合には、高水準言語を使用しても実行効率の良いマイクロプログラムの生成が困難になる。低レベル並列処理機構を実現している QA-2 の場合、レジスタ・ALU 部と同様に順序制御部のハードウェア構造も簡潔で分かり易くすることが、ユーザや高水準マイクロプログラム記述言語用コンパイラの負担を軽減するために重要である。

単純な順序制御方式の場合：

```
IF (P) THEN GOTO L1 ELSE NEXT;
IF (Q) THEN GOTO L2 ELSE NEXT;
IF (R) THEN GOTO L3 ELSE NEXT;
GOTO A;
L1:
IF (Q) THEN GOTO L4 ELSE NEXT;
   . . . . . .
L3:
CALL B;
   . . . . . .
L4:
IF (R) THEN GOTO L5 ELSE NEXT;
   . . . . . .
L5:
RETURN;
```

ＱＡ－２の場合：

```
CASE (P,Q,R) OF
        /000/ GOTO A
        /001/ CALL B
          . . . . . .
        /111/ RETURN;
```

図 4.4: 多方向分岐の効果

QA-2 の順序制御部 (SCU) では、その分岐形式として無条件分岐のほかに表 4.1の (1)(2) に示すような IF-THEN-ELSE 形式の 2 方向分岐および CASE-OF 形式の多方向分岐をハードウェア機能として装備している。また、表 4.1の (3)~(5) に示したように、条件分岐判定用論理関数はユーザが定義可能であり、その論理変数として最大 8 個のステータスを自由に組み合わせることができる。

これらの高度で複雑な条件判定や順序制御の機能を実現する制御論理は、高速アクセス可能でかつ書き換え可能な ECL・RAM チップによって実装されたテーブルを使用して実現している。この順序制御テーブルには、マイクロプログラム・アセンブラが翻訳したオブジェクト・データが実行に先立って格納される。ユーザは、制御記憶だけでなく、これらのテーブルを応用ごとに書き換えることができるので、条件分岐方式やステータスの制御方式をユーザ自身の応用に適した順序制御論理にすることができる。

QA-2 用マイクロプログラム・アセンブラを利用してマイクロプログラミングを行う際には、ユーザは各種順序制御テーブルの存在や容量をまったく意識する必要はなく、順序制御に関する煩雑なマイクロプログラミングから解放されている。高級言語風構文としての IF-THEN-ELSE 文や CASE-OF 文で記述されたマイクロプログラムの制御構造は、アセンブラが各種順序制御テーブルや制御記憶へのオブジェクト・プログラムとして翻訳し、編集する。これらの順序制御用ファシリティは主記憶をバックアップとして仮想化されているので、もし物理的容量を超えたオブジェクト・データであっても、SVP の管理下で動的にオブジェクト・データの入れ換えを行うことができる。この機能は、マイクロプログラムの制御記憶の書き換えのみが可能な従来のユニバーサル・ホスト計算機には装備されていない、大きな特長となっている。

4.3 順序制御部のハードウェア構成

SCU は 4K ワードの制御記憶 (CS) と、各種の制御テーブルを実装したマイクロプログラムの順序制御部とから構成されている。

4.3.1 マイクロ命令のフェッチ機構

マイクロプログラムの高速化を図るために、マイクロ命令のフェッチと実行をパイプライン化する手法がある。しかし通常、制御記憶へのアクセス時間は ALU 演算時間 (レジスタ参照時間を含む) と同程度のことが多いこと、マイクロプログラムには条件分岐が極めて多いこと、後続する命令で先行命令の結果を使うことが多い (データ依存性が高い) ことなどにより、マイクロプログラム制御計算機ではパイプラインのステージ数を余り増やしても効果があがらない [60]。従って QA-2 では、セーブド・モードとしてマイクロ命令のフェッチと実行という 2 ステージをパイプライン化した順序制御機構を採用した。

マイクロ命令 (図 4.2参照) の SCS フィールドによって、SCU におけるマイクロ命令のフェッ

項目	QA-2
(1)2 方向分岐形式	IF $f(A_0,\ldots,A_7)$ THEN { GOTO A / CALL B / RETURN } ELSE { GOTO C / CALL D / RETURN } ;
(2) 多方向分岐形式	最大 256 方向分岐が可能 CASE($A_0,\ldots,A_7$)OF /0/ { GOTO A / CALL B / RETURN } ⋮ /255/ { GOTO C / CALL D / RETURN } ;
(3)2 方向分岐 条件ステータス	最大 8 論理変数の自由な組み合わせ ($A_0,\ldots,A_7$)
(4)2 方向分岐 条件判定用論理 関数:f	ユーザが定義可能
(5) 多方向分岐 条件ステータス	最大 8 論理変数の自由な組み合わせ ($A_0,\ldots,A_7$)
間接分岐	"GOTO" あるいは "CALL" が可能
順序制御テーブル	2 方向分岐用 (1K エントリ)、 多方向分岐用 (4K エントリ)、 間接分岐用 (2K エントリ)
制御記憶容量	4K ワード/256 ビット
仮想制御記憶	デマンド・ページング方式 (1 ページ=512 ワード)
マイクロ・アドレス・スタック	512 レベル×2 本 (SVP による管理)
順序制御用マイクロ命令フィールド長	43 ビット

表 4.1: 順序制御部の仕様

チ・サイクルと RALU におけるマイクロ命令の実行サイクルとをオーバラップさせるか否かの選択を行う。セーブドモードでは、両サイクルがオーバラップし、順序制御に使用されるステータス情報は以前に実行されたマイクロ命令によるものが使用される。このモードでは、現在実行中の ALU 演算の結果を使用することはできないが、SCU と RALU との間でのステータス転送に伴うオーバヘッドは生じない。カレント・モードでは、両サイクルはオーバラップせずに、RALU による演算実行を待って SCU が次に実行すべきマイクロ命令アドレスの計算を始める。このモードでは、そのマイクロ命令による ALU 演算結果を直ちにそのマイクロ命令による順序制御に使用することができる。 両モードを使い分けることにより、RALU と SCU 間で複数ステータスの授受を同期させる際に生じる順序制御のオーバヘッドを、極力避けることが可能である。

4.3.2 ステータス・セーブの制御方式

QA-2 が 1 マイクロ命令の実行のたびに更新するステータス情報は 96 種類にものぼり、主なものとして ALU 演算結果によるものや MMP の操作結果によるものなどがある。これらのステータス情報は 256 ビットに展開され、さらにその各ビットごとにマイクロ命令の SMC フィールドによって指定された多様な選択操作 (save/set/reset/trigger/nop などの操作) を経て、ステータス・セーブ・レジスタ (SSR; 256 ビット) に設定される。(図 4.5参照) SMC フィールドはステータス・マスク・テーブル (SMT; (256 × 2) ビット × 256 エントリ) の 1 エントリを選択するための SMT アドレスを指定する。1 マイクロ命令の実行ごとに SMC フィールドに従って読み出される SMT の 1 エントリ (256 × 2 ビット) は、SSR の各 1 ビット情報に対する選択操作を 2 ビットで (4 種類のうちから) 指定する。また、SMT エントリの一部分は、ユーザが定義できるフリップ・フロップの操作、割り込みベクトルやマイクロ・アドレス・スタックのバンク選択、およびカウンタの増減操作などの指定にも使用しており、問題ごとに書き換え可能である。

4.3.3 マイクロプログラムの分岐形式

マイクロプログラムの分岐形式は、(A) 無条件分岐と、(B) 条件分岐 (2 方向分岐と多方向分岐) とに大別され、マイクロ命令 (図 4.2参照) の STY フィールドによって指定される。分岐形式のさらに細かい指定を行う ADG フィールドは、STY フィールドによる選択に応じて 3 形式のサブ・フィールド構成となる。ADG フィールドを使用して生成される順序制御情報としては、次に実行すべきマイクロ命令のアドレスの生成方法やアドレス情報の管理方法などがある。図 4.6に各種制御テーブルによる順序制御の流れを示した。

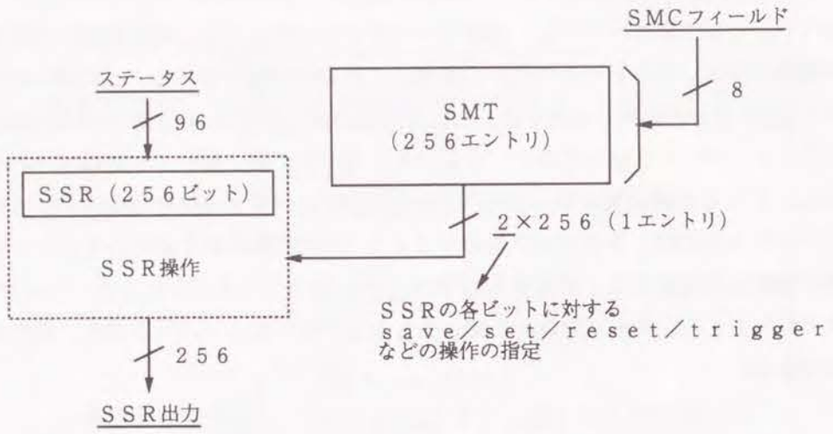


図 4.5: ステータス・セーブの制御方式

無条件分岐

無条件分岐が指定されている場合は、図 4.2に示したように、ADG フィールドは、さらに IDJ、ADS、DISP、ASS、ASD の各サブ・フィールドに分けられ、これらが直接順序制御情報として使用される (図 4.6の (1))。次に実行すべきマイクロ命令のアドレスは、基本的には DISP サブ・フィールドであるリテラル値と、ADS サブ・フィールドによって選択されたアドレス・セーブ・ファシリティ(マイクロプログラム・カウンタ:MPCR、割り込みアドレス・セーブ・レジスタ:ISVR、セーブ・レジスタ:SVR、アドレス・スタック: ASTK のいずれか) の内容とが加えられることにより生成される (図 4.6の (2))。例えば、ASTK が選択されるとマイクロ・サブルーチンからのリターン指定となる。ASS および ASD サブ・フィールドは、一度生成されたアドレス・データを退避したり、退避されていたアドレス・データを読み出したりする際に使用するアドレス・セーブ・ファシリティを選択する (図 4.6の (3))。例えば、ASD フィールドで ASTK が指定されると、マイクロ・サブルーチン・コールが実行されることになる。また、IDJ サブ・フィールドが “1” であれば、生成されていたデータをアドレスとしてさらに間接分岐テーブル (IJT; 2K エントリ) から読み出されたものが最終的なアドレス・データとなる間接分岐が行われる (図 4.6の (4))。

条件分岐

2 方向分岐や多方向分岐が指定された場合、図 4.2の ADG フィールドはテスト・ステータスの選択やテスト条件の設定を行うための各種順序制御テーブルのアドレス指定などに使用され、無条件分岐時のように ADG フィールドが直接順序制御情報の生成に使用されるわけではない。条件分岐の場合は、いくつかの順序制御テーブルが駆動されて初めて、無条件分岐時の

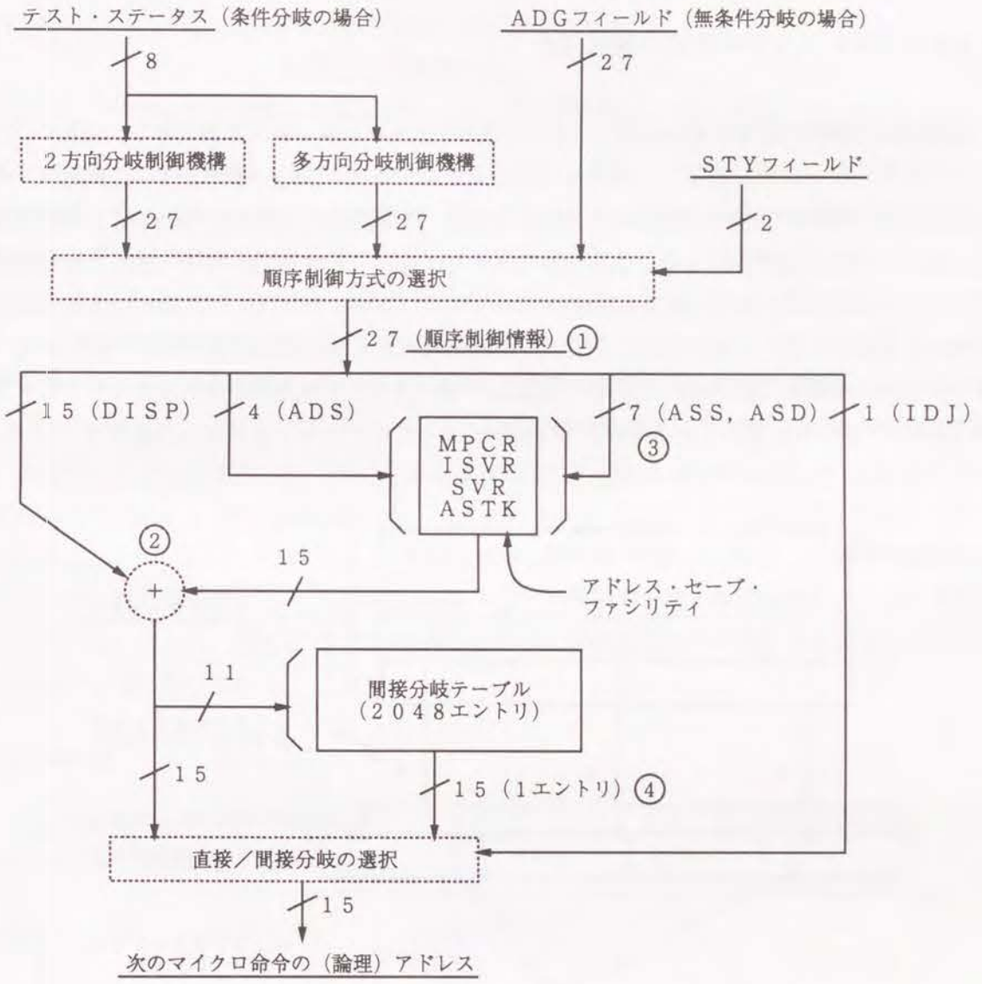


図 4.6: 順序制御の処理フロー

ADG フィールドと同じサブ・フィールド構成の順序制御情報が得られる (図 4.6の (1))。この順序制御情報を用いて最終的なアドレス・データが生成されるまでの過程は、無条件分岐の場合と同じである。次に、条件分岐における順序制御情報の生成過程について述べる。

4.3.4 テスト・ステータスの選択方式

条件分岐の場合の条件テストに使うテスト・ステータスは一度に最大 8 個であり、SSR の 256 ビットのうちから自由に選択し、組み合わせることができる。(図 4.7参照) 特に、低レベル並列処理機構 (複数個の ALU 演算) から発生する多種・多数のステータスを拾い上げ、順序制御用に使用できることが必要である。これらのテスト・ステータスはマイクロプログラムの条件文における論理変数: A_i (i=0,1,...,7) とみなすことができる。SSR の 1 ビットと A_iとの対応付けは、ステータス・ピック・テーブル (SPT; 256 エントリ) からマイクロ命令の SPC フィールドによって選ばれた 1 エントリによって行う。1 エントリには、8 個の 8 ビット・データが格納され、各々が SSR のアドレスを示している。

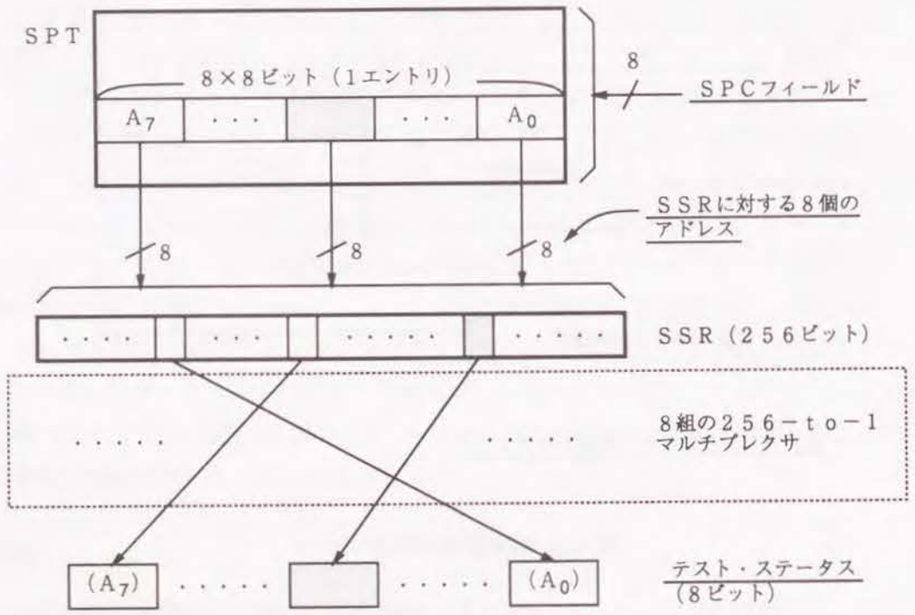


図 4.7: テスト・ステータスの選択方式

4.3.5 条件分岐の制御機構

2 方向分岐

マイクロプログラムの順序制御における 2 方向分岐は、高級マイクロプログラミング言語による記述形式である次のような "IF-THEN-ELSE" 文に対応するものである。

$IFf((A_0),(A_1),\ldots,(A_i))\ THEN$(分岐操作指定)
$ELSE$(分岐操作指定);
(ただし、fはユーザが定義した論理関数、(A_i) は論理変数 A_iの値である。$0 \leq i \leq 7$)

この文の THEN 節で指定する分岐操作と ELSE 節で指定する分岐操作にはいずれも無条件分岐と同様に GOTO 文、CALL 文、RETURN 文が何の制約もなく指定できるので、ユーザは構造化された制御構造を持つマイクロプログラムを記述することが可能である。

テスト条件である論理関数 fは、実際には真偽テーブル (TFT; 256 エントリ) のうち 1 エントリがマイクロ命令の IFJC サブ・フィールドによって選択される (図 4.8参照)。1 エントリは 256 ビットであり、テスト・ステータス (A_0)~(A_7) の値に対する真理値表となっている。

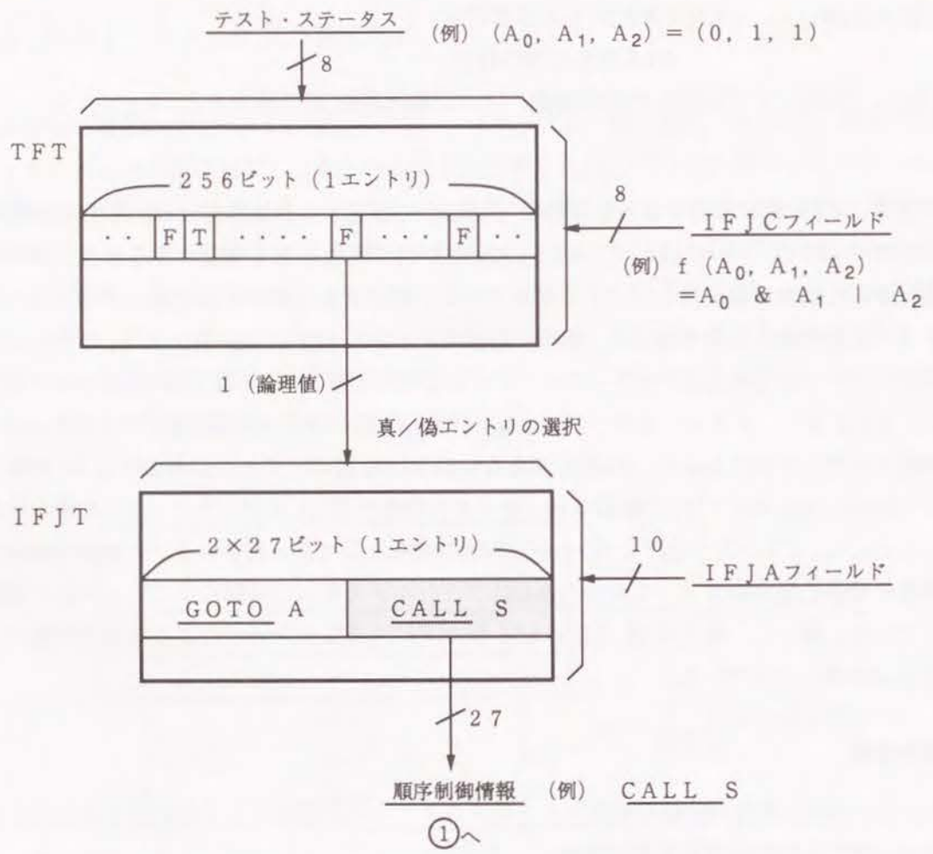
決定された fの値によって、THEN 節あるいは ELSE 節のいずれかに対応する IF 分岐テーブル (IFJT; 1024 エントリ) が駆動され、マイクロ命令の IFJA サブ・フィールドの値をそのアドレスとして、1 エントリ (27 ビット) が順序制御情報として読み出される。この順序制御情報は無条件分岐の場合のマイクロ命令の ADG フィールドとまったく同じサブフィールド構成となっている。従って、例えば IF 文に CALL 文や RETURN 文を混ぜたような高度な条件分岐の指定が可能となっている。

多方向分岐

QA-2 における多方向分岐の高級マイクロプログラミング言語による記述形式は次のような "CASE-OF" 文に対応するものである。

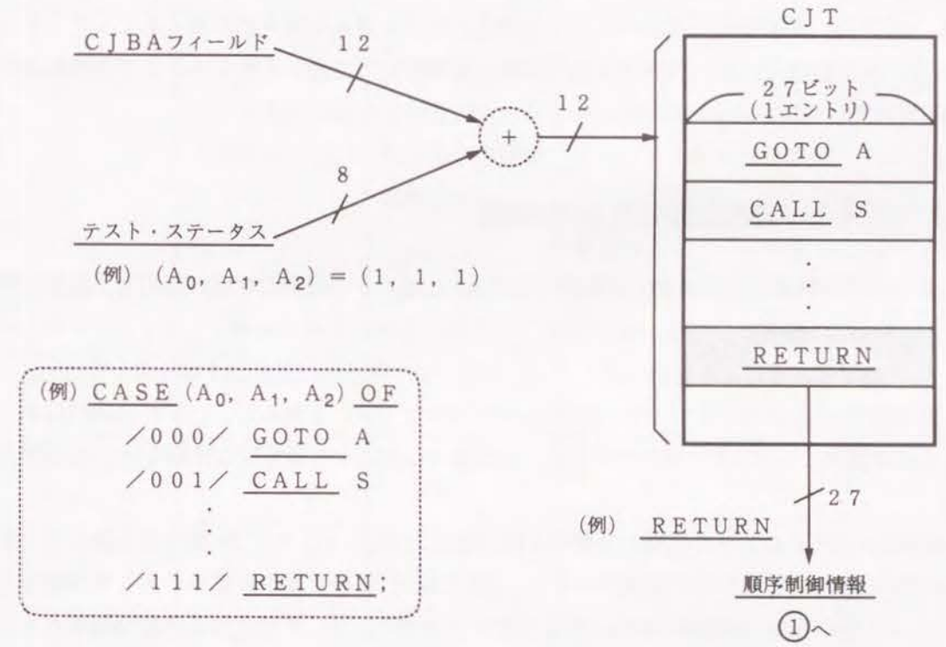
$CASE(A_0,A_1,\ldots,A_i)OF\ /0/$ (分岐操作指定)
$\vdots$
$/2^{i+1}-1/$ (分岐操作指定);
(ただし、A_iは論理変数 A_iの値である。$0 \leq i \leq 7$)

テスト条件である A_0~A_iは最大 8 ビットの論理値列であり、マイクロ命令の CJBA サブ・フィールドのリテラル値に加算されて、CASE 分岐テーブル (CJT; 4096 エントリ) の 1 エントリ (27 ビット) を選択する (図 4.9参照)。従って、CJBA サブ・フィールドの値は CJT のベース・アドレスであり、A_iの論理値列はそのディスプレースメントである。CJT から読み出され




```
(例)  IF  f (A0, A1, A2) =A0 & A1 | A2
      THEN GOTO A ELSE CALL S;
```

図 4.8: 2 方向分岐の制御機構




```
(例) CASE (A0, A1, A2) OF
     /000/ GOTO A
     /001/ CALL S
           .
           .
           .
     /111/ RETURN;
```

図 4.9: 多方向分岐の制御機構

るものは無条件分岐の場合の ADG フィールドと同じ順序制御情報であり、2 方向分岐と同様に多方向分岐の場合の分岐操作として CALL 文や RETURN 文も書くことができる。

4.3.6 仮想制御記憶方式

制御記憶 (CS) は 4K ビットの MOS スタティック RAM チップから成り、現在の実装容量は 4K ワードである。また、CS は 1 ページ=512 ワードでページ化され、15 ビットの論理アドレスによる仮想空間を構成している。CS のバックアップ記憶は主記憶 (MM) であり、QA-2 の機能部分の 1 つであるシステム管理プロセッサ SVP が、この仮想制御記憶方式を管理する。制御記憶や制御テーブルの仮想化により、大規模ユーザ・マイクロプログラミングが可能となっている。SVP は、仮想制御方式のページ入れ換えアルゴリズムを個々の問題に応じてダイナミックに変えることができる。CS や SCU 内の他の制御テーブルのダイナミックな切り換えの制御も SVP に任されている。

4.4 QA-2 の順序制御方式の評価

QA-2 の順序制御方式の有効性を確認するために、QA-2 を実際の問題に応用し、動的な性能評価を行った。例題としては、処理対象に明示的な並列性のある応用として、(1) スキャンライン・アルゴリズムによる 3 次元グラフィックス、また明示的に並列性を有しない応用として、(2) 逐次型 Prolog のインタプリタ、(3)Lisp のインタプリタ、を選んだ。いずれの応用においても、QA-2 のマイクロ・アーキテクチャについて様々な観点から定量的な評価を加えた。[55][44][72]

各応用における QA-2 の代表的な特性を抽出するために、(1) では 30 個の立方体をランダムに表示する場合の可視部分の決定ルーチン、(2) では 30 個の要素から成るリストを反転するプログラム、(3) では「tarai-4」のベンチマーク・プログラムを、それぞれ具体的な問題として選んだ。 まず、現状の順序制御部を "IF (X) THEN GOTO A ELSE NEXT ;" の単純な分岐形式 (1 個のステータスの値による 2 方向分岐) しか持たない構成にした場合に、各例題プログラムでどれだけの性能低下が生じるのかを調べた。表 4.2に示したように、3 次元グラフィックスや Prolog インタプリタでは、約 40%も性能が低下する。これらの応用に比べて Lisp インタプリタでは、Lisp の基本関数に対応するマイクロ・ルーチンをプログラム中に展開して埋めこむ方式によって高速化を図ったために、マイクロプログラムのモジュール化がなされず、24% の性能低下に留まっている。しかし、1 マイクロ命令当たりの平均 ALU 使用個数が多い Prolog インタプリタの場合には、それがおよそ 1ALU 分も低下する (平均 ALU 使用個数が約 3 個から約 2 個になる) ことが判明した。これは、ステータスを生成するレジスタ・ALU 部とそのステータスを使用する順序制御部との機能バランスが崩れて、低レベル並列処理機能が殺されてしまっていることを示している。

応用問題	現状の順序制御方式			現状の順序制御方式での総実行命令数に対する単純な順序制御方式にした場合†の総実行命令数の相対比
	総実行命令数に占める条件分岐命令の割合 (%)	総実行条件分岐命令数に占める 2 方向分岐命令の割合 (%)	総実行条件分岐命令数に占めるカレント・モード分岐命令の割合 (%)	
(1)3 次元グラフィックス	34	72	68	1.43
(2)Prolog インタプリタ	41	62	84	1.40
(3)Lisp インタプリタ	43	69	94	1.24

†" IF (X) THEN GOTO A ELSE NEXT ;" 形式の 2 方向分岐機能しかない場合

表 4.2: 順序制御部の動的評価

Prolog や Lisp のインタプリタは、個々のデータにタグ・フィールドを付加したデータ構造を基本とする直接実行型高級言語計算機のエミュレータとみなすことができる。これらの応用では、表 4.2からも分かるように、データそのものに明示的な並列性を持っている 3 次元グラフィックスよりも、条件分岐命令が出現する割合が多い。さらにこれらの応用においては、カレント・モード (そのマイクロ命令実行によって生成されたステータスを直ちに使用して順序制御を行うモード) による条件分岐の割合が多い (表 4.2参照)。このモードの分岐では、レジスタ・ALU 部による並列演算と SCU による順序制御機能がオーバラップされない。しかし、このモードによる分岐が多い理由は、生成されたステータスをセーブする必要がなく直ちに順序制御に使うからであり、QA-2 の並列演算機能と順序制御機能のいずれもが処理のあい路になることなく、機能バランスはとれていると言える。

タグ・アーキテクチャを採っている Prolog/Lisp マシンのエミュレーションの場合には、実行時に行われるタグの検査において、多数の (最大 8 個) ステータスによる多方向分岐機構が有効に機能している。またタグ・フィールドの各ビットは、既にそれぞれ独立した論理値としての意味を持っているので、マイクロプログラミング時にそれを論理変数として組み合わせ、条件判定用論理関数式とする操作は冗長で煩わしい。従って、これらの応用において 2 方向分岐の場合に使用された条件判定用論理関数にはあまり複雑なものは無い。2 方向分岐機能のハードウェア規模が割合大きいことを考慮すると、あまり利用されることのない 2 方向分岐機能のハードウェア化は得策ではない。従って、マイクロプログラム・コンパイラに論理関数の展開 (場合分け) を分担させて、ハードウェアとしては CASE-OF 分岐だけをサポートする構成方式も考えられる。(例えば、2 方向分岐命令 "IF X&Y THEN GOTO A ELSE NEXT ; " を、多方向分岐命令 "CASE (X,Y) OF /00/ NEXT /01/ NEXT /10/ NEXT /11/ GOTO A ; " に変換してしまう。) しかし、この方式ではディスパッチ・テーブルがすぐにあふれるおそれがある。ハードウェアとソフトウェアのトレードオフに関連する評価については、さらに考察を加える必要がある。

実測によると、QA-2 のマイクロ命令サイクルは、無条件分岐命令やセーブド・モードの条件分岐命令の場合には最小で約 600 ナノ秒である。この時、RALU における演算実行サイクルと SCU における順序制御サイクルはほぼ同じ時間を要し、完全にオーバラップしている。また、カレント・モードの条件分岐命令の場合には、RALU の演算実行サイクルが終了してから (ステータスが確定してから)、SCU の順序制御サイクルが始まるので、マイクロ命令サイクルは最小約 1200 ナノ秒となっている。

RALU と SCU とは 1 マイクロ命令サイクル中に 2 回、同期をとる必要がある。RALU から SCU へステータスを渡す場合と、SCU から RALU へマイクロ命令を渡す場合である。これらの同期はシェーク・ハンド方式で行っており、大規模ハードウェア構成による信号遅延とが合わさって、かなりのオーバヘッド (実測によると約 200 ナノ秒) となっている。ハードウェア実装方式を改良し、RALU と SCU を同一のクロックで制御する完全な同期制御方式を採用すれば、このオーバヘッドをなくすことができる。この場合にも RALU による低レベル並列処理機能と、SCU による順序制御機能との機能バランスはとれ、現在の QA-2 の方式をほとんど変更することなく適用できるものと考えている。

4.5 むすび

本論文では、QA-2 の順序制御方式の特徴と実現方式について述べた。QA-2 の応用としては、逐次型 Prolog マシンや LISP マシンのエミュレーション [55] [44]、および 3 次元図形表示システムへの応用 [72] などが既に行われている。種々の応用に対して、QA-2 の高機能順序制御方式が良く適合することを確かめている。

また我々は、高級言語 C で記述したプログラムを直接 QA-2 のマイクロプログラムへ変換するコンパイラを開発しており、その場合にも QA-2 で実現している構造化マイクロプログラミング方式が有効に機能している。例えば、条件分岐とマイクロ・サブルーチンとを機能的に組み合わせることが可能である点などは、コンパイル時における最適化戦略を構成するための有効な道具として利用されている。

現在[1]では、QA-2 のハードウェア実装時には入手できなかった種々の高機能ビルディング・ブロック (例えば、Weitek 社の浮動小数点演算器や AMD 社の Am293XX シリーズなど) が利用できるようになってきた。これらの LSI を有効に利用することによって、QA-2 は多数のユーザの要求を満たし得る次世代の高性能ワークステーション [59] となるであろう。

しかし、現状の QA-2 のハードウェア規模はかなり大きい (使用 IC 数は 22,000 個) ので、多様な応用実験を行って、ソフトウェア/ファームウェア/ハードウェアのトレードオフの見直しを図る必要もあろう。

[1] 1980 年代のことである。

Chapter 5

主記憶部の構成

5.1 まえがき

最近の半導体技術の進歩は、記憶素子に大きな影響を与えている。記憶素子の実装密度は、1年に2倍の割合で改善され続けているし、価格の低下も著しい。また、半導体記憶素子の実現によって、論理素子と容易に組み合わせることができるようになった。[3] しかし、計算機ユーザが主記憶に対して要求する、高速性と大容量という相反する特性を、同時に満たす素子はなく、計算機アーキテクチャのレベルで、両者のギャップを柔軟に吸収することが必要である。[11] QA-2では、QA-1における経験をもとに検討した結果、主記憶管理部(MMP)をCPUと独立に設計し、ユーザの要求に応えている。本章では、MMPの機能や特徴について、詳細に論じる。

5.2 MMPの概略

QA-1の主記憶は128k語(1語18ビット)の容量を持ち、4つのバンクから構成されていた。QA-1を各種の応用に適用した結果、主記憶の構成方式に関して、次の問題点が指摘された。

- MARが1つしかなく、アドレスの退避、復帰操作等のオーバヘットが大きい。そのために、個々のバンクを有効に利用できない。
- 最小アクセス単位が語(16ビット)であるため、バイト処理計算機をエミュレートする際のオーバヘッドが大きい。
- 主記憶参照時には、常にCPUフリーズが生じる方式なので、ALUの高速演算を生かせない。
- 主記憶に付加されているタグ・ビットが、レジスタ・レベルで処理できないので、その有効利用が図れない。

このような問題点は、マイクロプログラムを作成する際ユーザの大きな負担となり、主記憶の有効利用を妨げる原因となっていた。QA-2 は、RALU 部に強力なハードウェアを豊富に装備し、非常に柔軟な順序制御機能を実現して、RALU の演算能力、マイクロプログラムの生産性を飛躍的に向上させている。QA-2 を、各種の応用に対して効果的に適用していくには、これらの特長を最大限に利用できるよう、QA-1 の問題点を反省した上で、次の条件を満たす主記憶構成が必要である。

1. RALU における処理の高速性を妨げない。
2. 4 つの ALU のもつ並列性を充分反映する。
3. ユーザにバンクを意識させない、大容量一様空間である。
4. バイト単位で可変長データへのアクセスが可能である。

そこで、QA-2 の主記憶は、サイクルタイム 375ns の MOS-RAM(16k × 1 ビット) を用いて、1MB の実装空間を実現している。(実装空間は 16MB まで拡張可能であり、大容量ディスクをバック・アップにした仮想化を行えば、4GB の仮想空間を設定できる。) また、16 バンク×1 バイトの構成なので、大量データへの同時アクセスが可能である。MMP は、多方面からの主記憶参照要求を受け取り、上記 (1)~(4) の条件を満たしながら、それらを処理する。

5.3 MMP における処理の流れ

MMP は、表 5.1に示すように、RALU、SVP、VIDEO コントローラ、DMA チャネル、リフレッシュ回路の各部が発した 9 種の要求を処理する。MMP は、これら個々の装置が独立に発した要求を受け取ると、各装置との非同期動作によって、処理を開始する。複数の要求発生は、アービタ (arbiter) によって解決される。アービタは、複数個の要求の中で、最も優先順位の高いものを選択する回路で、同期式と非同期式がある [48]。MMP では、各要求発生源の非同期性や、MMP 内の基本クロックが他の装置に比べて遅く、同期式では要求受け付けまで、各装置がフリーズする時間が増加すること、などを考慮して、非同期アービタを採用している。その結果、MMP が要求待ちの状態にあれば、発生した要求を即時に実行できる。優先順位は、高速処理を必要とするもの (バースト・モード転送、DMA 転送) を高位に設定し、SVP の Read/Write のように、デバッグ時にのみ発生する要求を最下位にしている。優先順位の決定法は、単純な固定優先順位決定方式 (linear selection) である。アービタが 1 つの要求を受理した後に発生した全ての要求は、その処理が終了するまで、待ち (wait) 状態となる。MMP が、要求を受け付けた後の処理を、データの流れに従って概観すると、図 5.1のようになる。ここで示される通り、MMP の処理は、ノーマル・モードとバースト・モードの 2 つに大別することができる。

ノーマル・モードは、RALU、SVP、DMA チャネルによって管理される主記憶参照で、アクセス・アドレスからバンク内アドレスへの変換を MMP 内で処理し、必要に応じてアドレス

source	request
RALU	Read/Write Search Move
SVP	Read/Write CS change
Video controller	Video
DMA channel	DMA1 DMA2
refresh 回路	refresh

表 5.1: MMP への要求

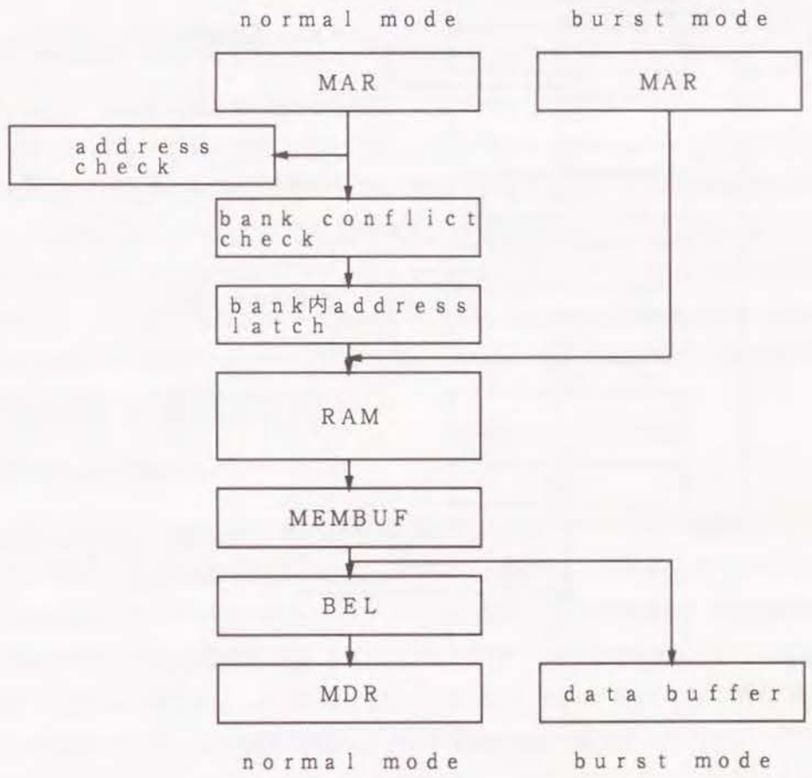


図 5.1: MMP におけるデータの流れ

の境界検査も行う。データに対しては、MDR と主記憶の間で、BEL によるデータ・アジャスト (5.4.5参照) を行う。一方、バースト・モードは、SCU、VIDEO コントローラによる主記憶参照である。アドレスは、常にバンク内アドレスで受け取り、データも 16 バイトの単位で転送するため、BEL も通過せず、その結果、非常に高速のデータ転送を実現できる。MMP 内の処理は、ノーマル・モードの場合 8 つのタイミング・パルス、バースト・モードの場合 5 つのタイミング・パルスで行う。内部クロックの周波数は 15 MHz なので、それぞれ約 550ns/約 350ns で処理を終了することになる。各タイミングにおける操作の概略を次に示す (図 5.2参照)。

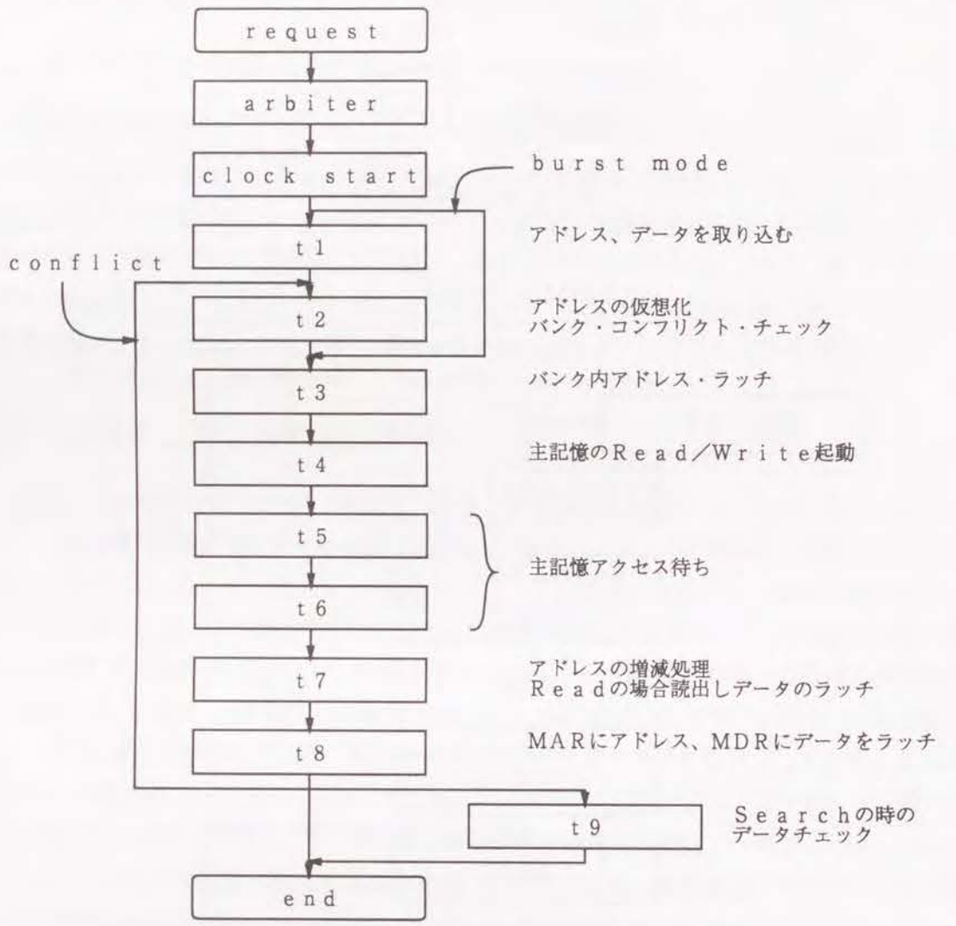


図 5.2: MMP における制御の流れ

t1 ノーマル・モードの各種アドレス、データを取り込む。

t2 アドレスの変換を行い、バンク・コンフリクトをチェックする。Write の場合、データ・アジャストを行う。

t3 バンク内アドレスをバンク毎にラッチする。バースト・モードのアドレスを取り込む。Write の場合、データを MEMBUF にラッチする。

t4 主記憶への Read/Write を開始する。

t5〜t6 主記憶の Read/Write 終了待ち状態である。

t7 アドレスの増減処理を行う。Read の場合、データを MEMBUF にラッチする。

t8 増減したアドレスを MAR にラッチする。Read の場合、データを MDR にラッチする。

t9 Search の場合、データ・チェックを行う。

バースト・モードで使用するのは、t3〜t7 である。次節から、各装置の要求処理方法と、それぞれに対応した、MMP の機能を合わせて詳述する。

5.4 RALU の要求処理

QA-2 の RALU では、ALU 周辺のレジスタ・ファイルを大容量化し、各種のテーブルなどを、レジスタ・レベルで管理することができる。しかし、大量データや、入出力データなどは、やはり、主記憶上に格納せざるを得ない。特に高級言語処理では、中間言語の系列を主記憶に持つ手法が一般的である。そこで、主記憶管理方式の設計では、RALU の要求を、柔軟かつ高速に処理することを、第一の目標としている。MMP は、RALU との間に多数の信号線を設けて、通信専用レジスタ間で高速のデータ転送を行っている。そして、頻繁に生じる RALU の主記憶参照を、効率良く処理するための機能を備えている。本節では、RALU の要求処理を、各機能単位毎に説明していく。

5.4.1 命令のデコード

RALU からの処理要求は、マイクロ命令の MMC,MAM フィールドの設定によって行われる。RALU では、マイクロ命令のフェッチと同時に、上記フィールド転送パルスを MMP に出力する。MMP は、そのパルスを利用してフィールドを取り込み、デコードを開始する。MMC は、MMP に対する要求の種類を指示するフィールドで、1)Read/Write 2)Search 3)Move のいずれかを指定する (表 5.2)。MAM は、MMC で Read/Write を指定した時に意味を持ち、Read,Write の別とアドレスの増減を定める。MMC のデコード結果は、RALU 内の処理の正常終了を示すパルスによって効力を持つ。RALU 内で、レジスタ・ファイルのアクセス等でエラーが生じた場合、MAR,MDR の値が保証されないので、RALU は無効パルスを発し、MMP の起動を中止する。次に、1)〜3) の機能と処理について詳述する。

MMC	operation
00	No Operation
01	Read/Write
10	Search
11	Move

MAM	function
000	No Operation
001	Read
010	Read & Increment
011	Read & Decrement
100	Write
101	Write & Increment
110	Write & Decrement
111	

表 5.2: MMC,MAM フィールドの意味

1) Read/Write:主記憶に対する読み出し、書き込み要求であり、次の 4 種類のレジスタを使用する。

MAR(Memory Address Register) 32 ビット Read/Write すべきデータの先頭アドレスを示す。

MRWCR(Memory Read Write Count Register) 4 ビット Read/Write すべきデータの長さをバイトで示す。実際には、この値に 1 加えた長さを Read/ Write する。

MAIDCR(Memory Address Inc/Dec Count Register) 4 ビット Read/Write 終了後のアドレスの増減値を示す。実際には、この値に 1 加えた値を増減する。

MDR(Memory Data Register) 32 ビット Read したデータ、あるいは、Write すべきデータを示す。

RALU 内には、これらのレジスタが、スペシャル・レジスタ内に 4 個づつ装備され、全ての ALU から自由にアクセスすることができる。MMP への要求は、これら 4 個のレジスタを 1 組として行い、MAM のフィールド番号と、各レジスタ番号が対応している。即ち、MAMi の操作要求の処理には、固定したレジスタ (レジスタ番号 i)4 種を利用する。これらのレジスタは、RALU とは別に MMP 内にも装備され、内部クロックがスタートした最初のタイミングで、RALU 内のデータを MMP に取り込む。この後、RALU と MMP は各々の処理を非同期に行う。MMP で使用しているレジスタは、RALU 側でアクセス不能になり、もし、そのレジスタへのアクセス要求が出されると、RALU はそこでフリーズし、MMP の処理の終了を待つ。RALU は、4 つの MAM フィールドで、それぞれ独立に Read/Write を指定できる。MMP 内では、Read を処理した後、Write を処理するので、RALU への Read/Write 処理の終了信号 (レジスタのアクセス許可信号) は、1 回の Read/Write 終了毎に出力している。

このように、ユーザは 4 種のレジスタに値を設定するだけで良く、あとは、全てハードウェア的に主記憶アクセスを行う。MMP の起動による RALU のフリーズも、レジスタ・アクセス時のみに留まっており、ここに、非同期動作方式の有効性をみることができる。さらに、RALU からの連続アクセス要求に備えて、MAM フィールドの値のみを、MMP 内キューに保存する機能を有している。キューは、図 5.3のように 12 ビット × 16 の大きさを持ち、MMP 動作中に RALU が Read/Write を要求すると、その命令を格納する。キューが空になるまでキュー内の命令によって主記憶アクセスを繰り返す。キューが一杯になった時に初めて、RALU はフリーズする。

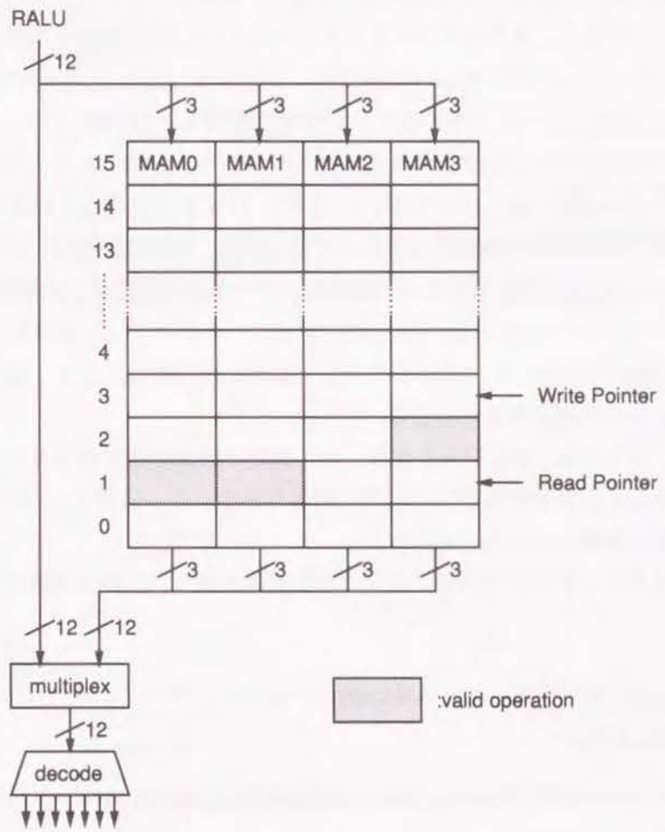


図 5.3: キューの構成

2) Search:32 ビットデータの検索機能であり、次に示す 5 種類のレジスタを利用する。

MSTR(Memory Search starT register) 32 ビットデータ検索の開始アドレスを示す。

MSPR(Memory Search stoP register) 32 ビットデータ検索の終了アドレスを示す。

MSCR(Menory Search Count Register) 32 ビット MSTR の増加値を示す。1 回の比較後、MSTR、MSCR の値の和のアドレスが次の検索アドレスとなる。

MSDR(Memory Search Data Register) 32 ビット検索データを示す。

MSMR(Memory Search Mask Register) 32 ビットデータのマスク値を示す。MSDR の内容と論理積をとった値が、実際の検索データとなる。

Search は、主記憶の全アドレス空間に対して、32 ビットまでの任意のデータを、RALU からの Read/Write と並行して検索する機能である。プログラムによる表の探索や、文字列の走査は、比較的時間がかかるうえ、その間他の主記憶アクセスを行えない。これをハードウェアでサポートすることにより、高速化を図ることができる。Search の処理手順は、まず、MSTR の示すアドレスに対して、通常の Read 操作を行い、データを MDR1 に格納する (図 5.4、5.5)。この後、MDR1、MSDR にマスクをかけて、両者を比較する。MSMR のマスク値によって、任意の部分データを抽出できるので、あらゆる長さやパターンを持つデータの検索が可能である。比較の結果、データが一致しなければ、MSTR を MSCR だけ増して検索を続ける。これによって、表のエントリ単位での検索が容易に実現出来る。探索は、指定したデータを発見するか、アドレスが MSPR の値を越えるまで続ける。データ発見時には、検索正常終了として、RALU 内のフラグをセットする。この時、MSTR がデータのアドレスを保持している。アドレスが MSPR を越えた場合には、検索終了を割り込みで知らせる。こうした終了報告機能によって、プログラムの処理を簡単化できる。

Search の処理では、Read 後にデータ比較、終了判定の時間を要するので、タイミングパルスを 9 つ利用する。1 回の検索毎にアービタに制御を戻すので、RALU の Read/Write など、より優先順位の高い処理を受け付けられる。

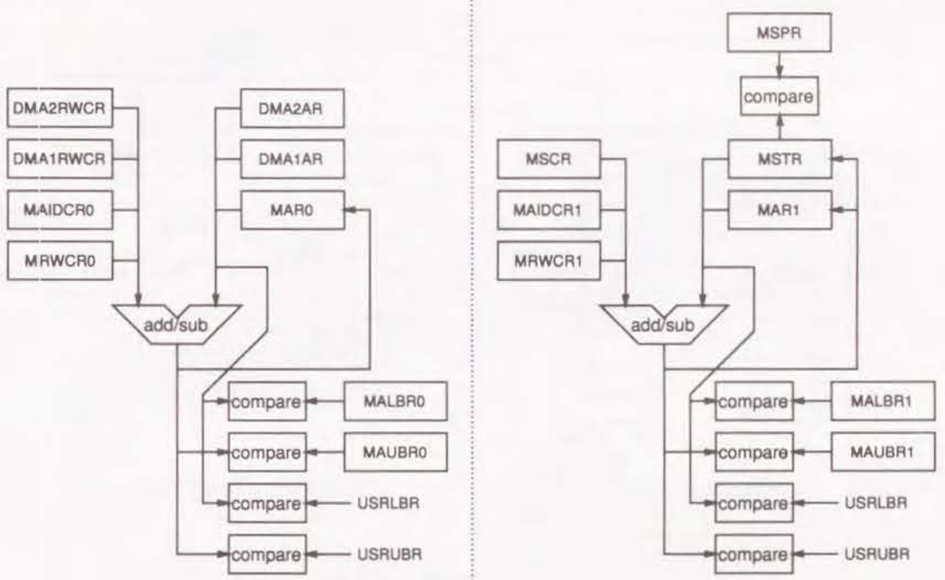
3) Move:ある領域から他の領域へのブロック転送機能であり、次の 3 種類のレジスタを使用する。

MVSAR(Memory moVe Source Address Register) 32 ビットソースブロックの転送開始アドレスを示す。

MVDAR(Memory moVe Destination Address Register) 32 ビット転送先ブロックの格納開始アドレスを示す。

MVCR(Memory moVe Count Register) 32 ビット転送バイト数を示す。

プログラムを作成する際、あるデータのコピーを作ったり、ブロック単位のデータ転送を行うことは、比較的よく行われる処理である。そして、そのほとんどは、他の演算処理と並行して行うことができる。MMP では、ブロック転送をハードウェア化することによって、RALU における、処理の効率化を図っている。ブロック転送において注意すべき事柄は、データの破



(a) MAR0周辺　(b) MAR1周辺

(c) MAR2周辺　(d) MAR3周辺

図 5.4: MAR の構成

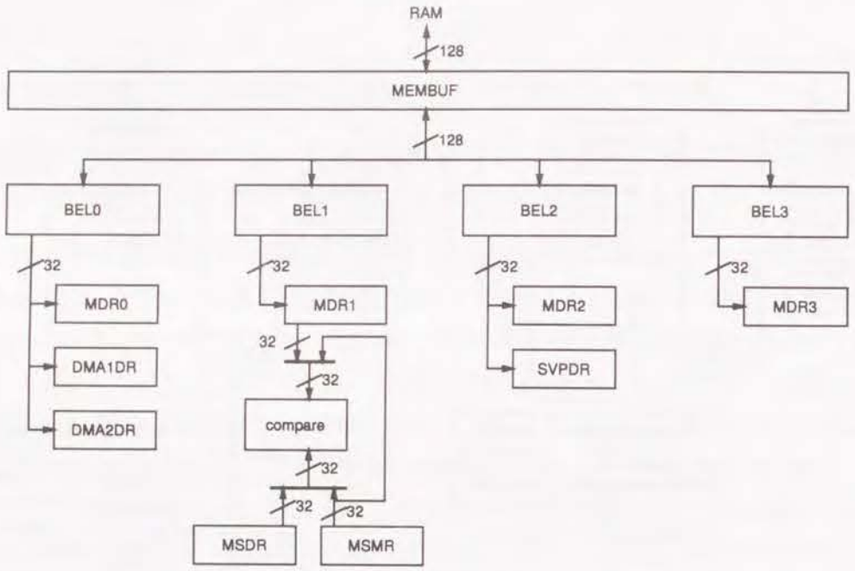


図 5.5: MDR の構成

壊を防ぐことである。つまり、(図 5.6(b)) に示すように、転送領域と転送先領域が重なり合うとき、先頭データからの転送は、自らのデータ損壊を招いてしまう。そこで、MMP では、データ転送の前に、MVSAR と MVDAR を比較し、MVSAR > MVDAR ならば先頭のデータから (図 5.6(a))、MVSAR < MVDAR ならば最後のデータから転送を開始する (図 5.6(b))。ブロック転送要求を受け取った後、最初のサイクルで上述の判定と、必要に応じてアドレスの変更を行い、次のサイクルから実際の転送処理に移る。転送には Read、Write2 つのサイクルが必要で、共に 16 バイトのバス幅を活用して、転送の高速化を図っている。転送の終了は、MVCR=0 によって判定し、RALU に知らせる。ブロック転送の場合もデータ検索と同様に、1 回の転送 (2 サイクル終了) 毎に、アービタを通し、他の要求の受け付けを行う。

5.4.2 MAR と MDR

MAR は 4 個存在し、ユーザからみて、それらは全て同一機能を有する。MAR は 32 ビットであるが、現アドレス空間は 1MB なので、そのうち 20 ビットのみが意味をもつ。上位ビットは、実装容量の拡張 (16MB まで)、外部ディスクとの仮想化 (4GB まで) の際に用いる。MAR の周辺は、図 5.4(a)〜(d) に示すように、加減算器を中心とした、各レジスタ類と、アドレス・チェック用比較器によって構成している。各種レジスタは、RALU や他の装置から初期値を取り込む。加減算器では、32 ビットの演算を行い、その右入力は各種のアドレス・データ、左入力はデータ長、あるいは、アドレスの増減値である。ここでは、アドレスとデータ長の加算に

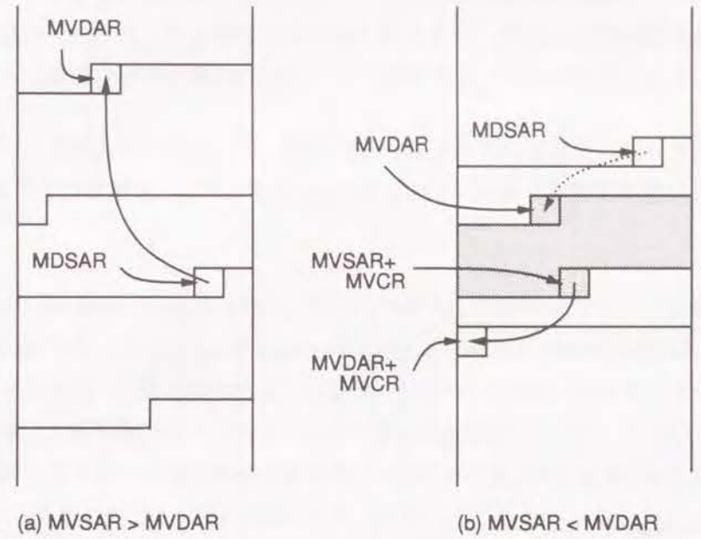


図 5.6: ブロック転送の例

よって、アドレス・チェック (5.4.3参照) 用の値を生成すると共に、アドレスの増減指定があった場合、指示に従って次のアドレス値を求める。この増減機能により、ユーザは Read/Write を指定するだけで、アドレスが連続するデータへのアクセスを容易に行うことができる。

MDR は、MAR に対応してやはり 4 個存在し、MARi のアドレスに Read/Write するデータには、それぞれ MDRi が対応している。図 5.5に MDR の構成を示す。MDR を各々1 つの BEL (5.4.5参照) と接続し、Read の場合のマルチプレクス (multiplex)、Write の場合のデマルチプレクス (demultiplex) 処理を行う。Read/Write するデータ長の指定は、4 ビットで行うので、1 つの MAR の示すアドレスに、最大 16 バイト・データを Read/write できる。この場合、1 つの MDR(4 バイト) に納まりきらないので、対応する MDR と、それに続く番号をもつ MDR にまたがって、データを保持する。

図 5.4、5.5から分かる通り、normal mode による主記憶アクセスでは、アドレスやデータが 4 つの MAR、MDR とバスを共有する構成をとっている。

5.4.3 アドレス・チェック

プログラムの実行時エラーには、変数に不当な値がセットされ、プログラムが暴走したり、データ領域やプログラム領域を破壊したりすることに起因するものが多い。こうしたエラーは、変数の値を逐次検査することによって防ぐことができる。特に、高級言語の場合には、変数の管理はその処理系が受け持つので、処理系作成時における、確実なエラー対策が必要とされる。

しかし、プログラムを作成する際、アドレス・チェックは非常に繁雑な作業であり、かつ、プログラムの構造を複雑にするので、プログラム実行時におけるオーバヘッドの要因となる。一方、アドレス・チェックをハードウェアで行うと、比較的容易に回路を実現できる。

そこで、MMP では、主記憶にアクセスする時点で、アドレスの正当性を、ハードウェア的に確認する、主記憶保護機構を備えている。図 5.4に示すように、2 種類のアドレス・チェック機能がある。

(1) MAR 単位のチェック:ユーザが、あらかじめ MAUBR(Memory Address Upper Boundary Register)、MALBR(Memory Address Lower Boundary Register) に、その MAR のアクセス上・下限値を設定しておく。RALU からの主記憶参照要求が来る度に、これらのレジスタの値とアドレスを比較する。この時、MALBR はアクセス・データの先頭番地と比較し、MAUBR はデータの最終番地 (先頭番地+データ長) と比較する。その結果、いずれかが境界値を越えていれば、アドレス・エラーとする。ユーザは、エラー発生を RALU 内フラグによって、MAR 毎に認識し、エラー処理に利用することができる。このアドレス・エラーについては、ユーザの設定できるマスクがあり、対応するマスクを "1" にすると、エラー発生時にはそのアドレスへの Read/Write を中止する。一方、マスクを "0" にしておけば、エラーが発生しても、RALU 内のフラグを設定するだけで、Read/Write 処理は続行する (図 5.7)。

(2) ユーザ単位のチェック:QA-2 は、制御記憶や順序制御テーブルのバックアップとして、主記憶を利用しているため、主記憶の一部はシステム領域として、ユーザのアクセスを禁止している。この領域が不当なユーザプログラムによって破壊されるのを防ぐために、MAR 単位に設定する上・下限値とは別に、個々のユーザに対して境界値を設定し、アドレスのチェックを行なう。この境界値は SVP によって決定され、制御記憶にマイクロプログラムをロードする際、USRUBR(USeR Upper Boundary Register)、USRLBR(USeR Lower Boundary Register) に上・下限値がそれぞれ設定される。ユーザは、この値の参照、変更を許されない。エラー発生時には、RALU、SVP 双方に伝え、SVP が RALU の演算を停止する。このアドレスエラーに対しても、マスクを備えているが、主記憶内にシステム領域が存在する時には、強制的に "1" とする。

このように、ハードウェアでアドレスチェック機能を有することは、不当な命令フェッチや、不当なデータアクセスを事前に自動検出できるので、プログラムとデータを明確に区別することにつながり、プログラム保護、システム保護に非常に有効な手段である。

MAR 単位のチェック機能は、RALU からの Read/Write 要求に対してのみ有効で、ユーザ単位のチェックは、RALU からの要求すべてに対して効力を持つ。その他の要求に対するアドレスチェックは行なわない。

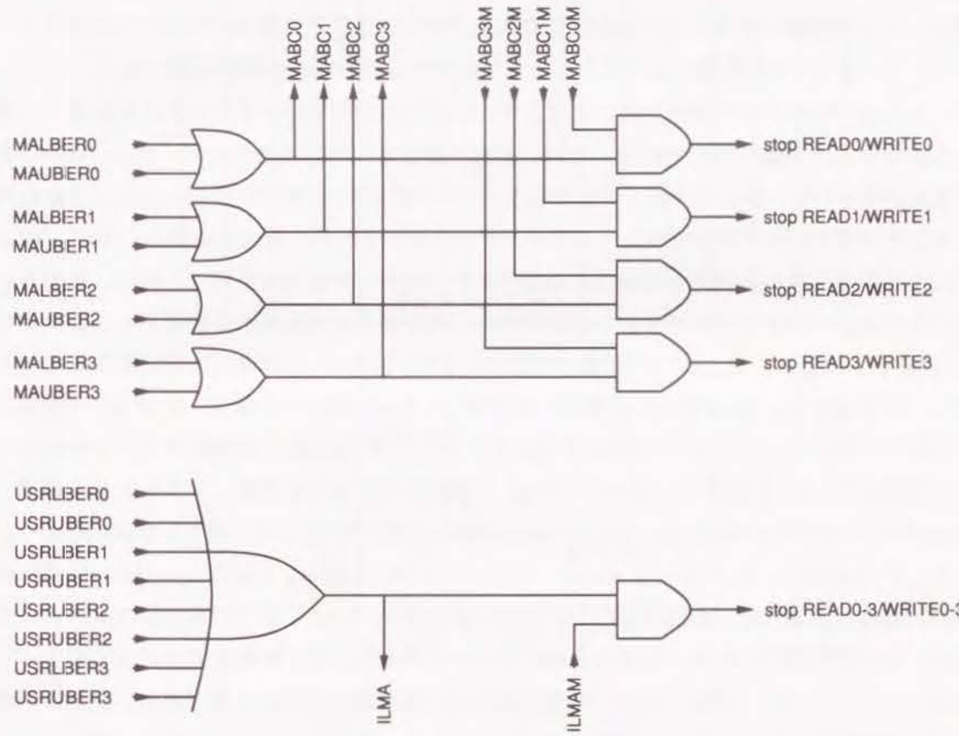


図 5.7: アドレス・エラーの処理

5.4.4 コンフリクト・チェック

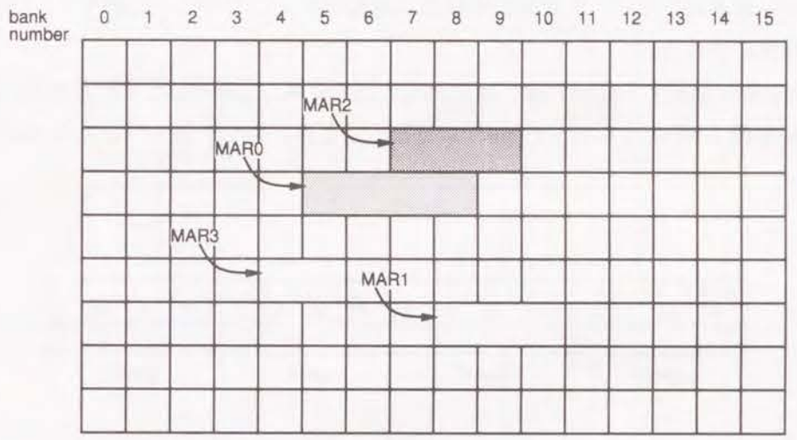
RALU からの Read/Write 要求では、ユーザは 4 個の MAR を利用して、全く異なる 4 つのアドレスへのアクセスを要求することができる。各 MAR へのアドレス設定は、バンク構成を意識することなく行えるので、アクセスするデータが、主記憶内でどのような位置関係にあるかは、全く不定である。ところが、主記憶は 16 バンク構成だから、1 回の Read/Write 操作でアクセスできるのは、各バンク 1 バイトずつ、計 16 バイトである。例えば図 5.8(a) のような場合、バンク 5、7、8、9 にそれぞれデータの重なりがあり、1 回の Read/Write では全てのデータにアクセスできない。図 5.8(a) ではバンク 8 に 3 つのデータが含まれるので、最低 3 回の主記憶アクセスを必要とする。このように、同一バンクへのアクセス要求が発生する状態 (バンク・コンフリクト) は、頻繁に起こると考えられるので、その高速処理が必要である。

バンク・コンフリクトの解決法は、まず、各 MAR で Raed/Write するバンクを計算し、それらの重なり具合を調べる。その後、同時に処理可能なものを組み合わせて、Read/Write を繰り返す。即ち、アドレスの下位 4 ビット (バンク・アドレス) とデータ長から、各 MAR の対象とするすべてのバンクを、ビット・パターンで求める。この時、高速化を図るために、ECL 素子による並列シフタを利用している。このビット・パターンを MAR 毎に比較し、重なりの有無を求める。バンク・コンフリクトを認識すると次のように処理順序を決定する。優先順位を MAR0 > MAR1 > MAR2 > MAR3 と定め、まず、MAR0 と同時に処理できるものから選択し、以下 MAR1、2、3 について同様に処理する。例えば図 5.8の場合、各 MAR の要求に (b) で示すバンク・コンフリクトが存在する。そこで、まず MAR0 と同時にアクセスできるもの (この場合はなし) を処理し、続いて MAR1、MAR3 のデータを同時にアクセスし、最後に MAR2 のデータにアクセスする。全ての Read/Write が終了するまで、図 3.2 で示したようにタイミングパルスは、t8→t2 とフィード・バック・ループを回る。1 回の Read/Write 操作終了毎に、処理した MAR、MDR を RALU に知らせるので、RALU 側での各レジスタへのアクセスが、その都度許可される。全ての Read/Write を終えたとき、制御をアービタに戻す。

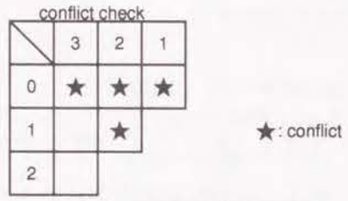
バンク・コンフリクトが生じると、処理時間の最大値は最小値の約 4 倍である。MMP 起動から終了までのアクセス時間が、プログラム全体に大きく影響を及ぼす場合には、データの主記憶内割り付けを、十分考慮する必要がある。一方、その必要性がなければ、ユーザは主記憶本体を全く意識せずとも良く、マイクロプログラム作成上の負担軽減に大きく寄与すると考えられる。

5.4.5 BEL

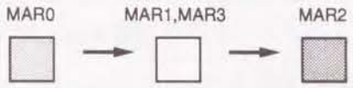
主記憶は 16 バンクから構成され、1 回のアクセスで 16 バイトまでの Read/Write が可能であるのに対し、MDR は各々4 バイトだから、Read の場合には 16 バイトの中から必要な 4 バイトを抽出する操作が必要である。また、Write の場合には、4 バイトのデータを全てのバンクに書き込める様に処理する。この機能を果たすのが、BEL(Bus Exchange Logic) と呼ばれる



（a）データの主記憶内位置



（b）コンフリクトの状況



（c）処理順序

図 5.8: バンク・コンフリクトの例

data adjust 機構である。QA-2 では 128 バイトとバス幅が大きく、同時に 32 ビット×4 のデータを処理するので、高速かつ簡潔な回路による実現が必要である。次に、MMP で用いた回路の原理を説明する。

Read の場合 (multiplex)

主記憶から読み出したデータを MEMBUF に格納する。このあと、処理はバイトの抽出とシフトの 2 段階に別れる。

1 つの MDR に選択するデータは、MEMBUF 内で連続した高々4 バイトである。なぜなら、MEMBUF を図 5.9のように 4 つの port に分割したとき、データが 3 つ以上の port に渡ることはない。

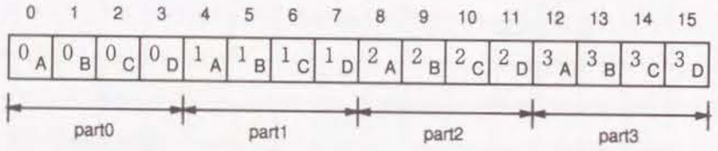


図 5.9: MEMBUF

図 5.10を利用して、データの抽出を説明する。

(a) は アドレスの下位 4 ビット (a3 a2 a1 a0) : 0010
　　データの長さ　　(l1 l0)　　: 10

とした時のデータ抽出である。つまり、(a3 a2)=00 だからデータの先頭が port 0 にあり、(a1 a0)=10 によってその 3 バイト目から始まることが分かる。このように、アドレスの下位 4 ビットとデータ長から求めるデータの位置を決定できるので、各 port の添字 (A～D) と MDR のバイト名 (A～D) を一致させるよう、バイト単位に取り出してくれば良い。(b) は (a3 a2 a1 a0)=1011、(l1 l0)=01 とした場合の処理を示している。

こうして抽出したデータは、MDR 内で正しい順序に並んでいないので、次にこれをシフトすることによって、正しいデータを得る。このシフト数も、アドレス下位 2 ビットとデータ長から簡単に求めることができる。

Write の場合 (demultiplex)

Read の場合と全く逆操作である。即ち、MDR のデータを、まずシフト (Read の場合の逆方向シフト) した後、1 つのバイトを 4 つの port のいずれかに転送することによって実現している (図 5.11)。

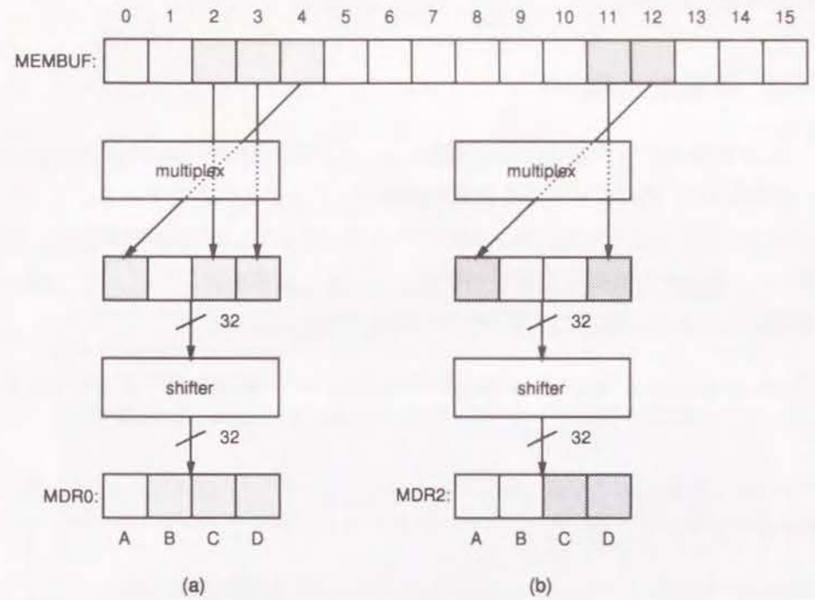


図 5.10: BEL の処理方式 (Read)

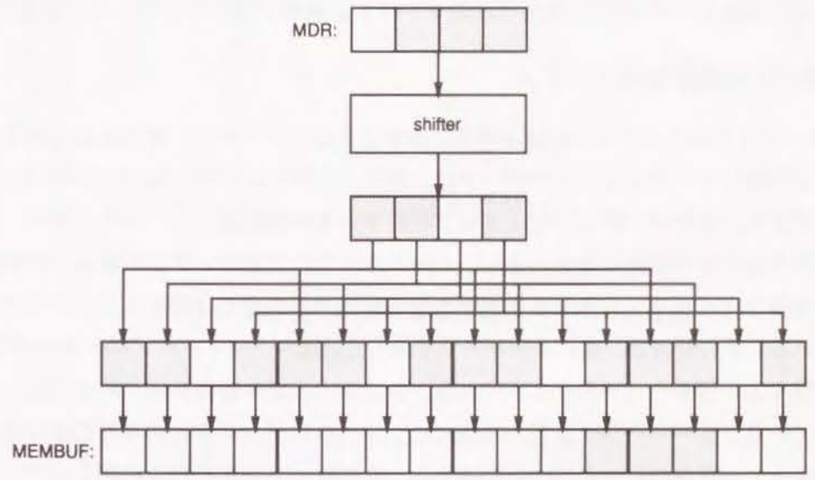


図 5.11: BEL の処理方式 (Write)

BEL は normal mode では必ず通過する回路であり、MMP では高速化を図るために、ECL 並列シフタを利用している。

5.5 SCU の要求処理

MMP と SCU の間のデータ転送には 2 種類ある。1 つは制御記憶 (control storage) の入れ換えであり、他方は SCU 内に存在する各種順序制御テーブルの入れ換えである。もっとも、両者の転送方法は全く同じ方式をとり、転送開始アドレスによって、SCU が判断する。転送の起動は SVP が行い、転送中の制御は MMP が管理している。転送起動に先立って、SVP は MMP 内の次の各種レジスタに、転送に必要な制御情報を設定しておく。

CSAR(Control Storage Address Register):16 ビット 転送開始アドレスを示す。実際には 20 ビットのレジスタで、上位 4 ビットは外部スイッチにより設定する。

CSMR(Control Storage Mask Register):8 ビット MMP、SCU における、Write 操作時のデータ・マスク値。各ビットで 2 バイトづづマスクする。

CSCR(Control Storage Count Register):16 ビット 転送回数を示す。

転送が起動されると、MMP はまず CSAR、CSMR を SCU に転送する。この制御情報を利用して、SCU は転送データを認識し、受け取り準備をする。次のサイクルで、データの転送を開始する。転送データは、あらかじめ DMA 転送によって、外部装置から主記憶内に格納されていなければならない。SCU とのデータ転送は、burst mode によっておこなうので、約 40MB/s の転送速度が保証される。SCU 内各種制御テーブル、制御記憶について、次に説明する。

5.5.1 SCU 内順序制御テーブル

SCU は、マイクロプログラムの順序制御に関する大量のデータを、書き換え可能な制御テーブルの中に記録している。これらのデータは、各マイクロプログラム毎に、アセンブラが作成したものである。そこで、新しいマイクロプログラムを制御記憶にロードする度に、表内の情報を書き換えることが必要になる。また、ハードウェアによるテーブルの容量には制限があるので、大規模なマイクロプログラムに対する順序制御情報を全て保持することができない場合も考えられる。その時には、表のオーバーレイを行う。即ち、マイクロプログラムの実行中に、必要な情報を主記憶からロードする訳である。主記憶内の順序制御テーブルは、図 5.12の様に配置されている。オーバーレイを行う場合には、マイクロプログラムの実行回復を早めるためにも、必要なテーブルのデータだけを転送する。この時、CSMR の値を利用してデータの切り出しを行っている。

順序制御テーブルとのデータ転送路は双方向であり、テーブル内の値を主記憶に保存する処理も、この転送路を用いて行うことができる。

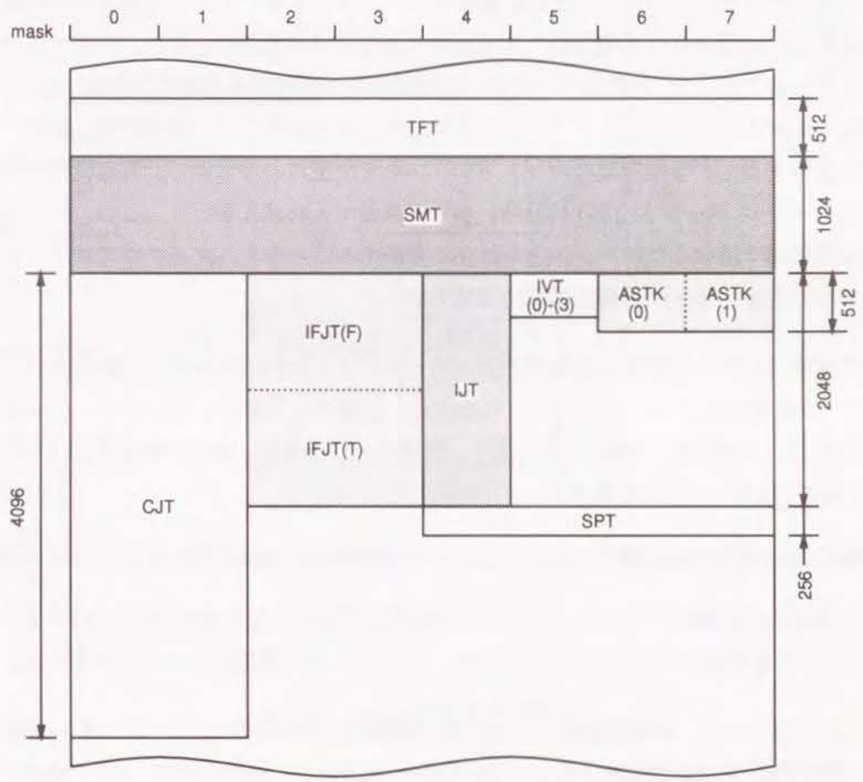


図 5.12: 主記憶内における順序制御テーブルの配置

このように、ユーザ毎に順序制御テーブルを用意し、その仮想化を実現することは、ユーザの各問題に対して QA-2 が効果的に適応できることを示すものである。

5.5.2 制御記憶部

制御記憶部は、SCU 内の 1 つのモジュールであり、制御記憶と仮想化回路から構成される。制御記憶は、物理アドレス空間が 4k 語 (1 語:256 ビット) で、アクセスタイム 45ns の MOSRAM(4k×1 ビット) を用いている。書き換え可能な制御記憶 (Writable Control Storage) を利用した、アーキテクチャの可変性が重要視され、マイクロプログラムの使用が急激に増加している今日、4k 語という容量は決して充分な大きさとは言えない。また、多重マイクロプログラミングをおこなって、複数のユーザが、制御記憶を分割利用する場合を想定すると、当然、その容量不足が予想される。そこで、仮想制御記憶方式を採用して、制御記憶の論理アドレス空間 32k 語を実現し、仮想空間のバック・アップを主記憶にとっている。仮想化は、制御記憶を 8 つのページ (1 ページ 512 語) に分け、ページ単位で入れ換えを行う。この、ページの概念は、仮想化処理で用いるだけで、ユーザは全く意識することなく、マイクロプログラムを作成することができる。以下に制御記憶部の機能を示す。

アドレス変換 SCU から渡される論理アドレス (15 ビット) を、論理ページ番号 (6 ビット) とページ内相対アドレス (9 ビット) に分ける。論理ページ番号によってページ・テーブルを索引し、物理ページ番号を読み出す。物理ページ番号と、相対アドレスにより、物理アドレス (12 ビット) を生成する。

制御記憶の Read/Write 物理アドレスによって制御記憶を Read/Write し、SCU に渡す。

ページ・フォルトの検出 ページ・テーブル内に対応するページが無いとき、SVP にページ・スワップを依頼する。リプレース・ページは FINUFO(後述) によって決定する。

図 5.13に示すように、制御記憶の入れ換えは、制御記憶部からのページ・フォルト情報を SVP に伝え、その後、MMP が起動される。制御記憶の転送中、CSMR は全て “1” である。1 ページの転送には約 400μs 要する。

リプレースメント・アルゴリズム

制御記憶部では、FINUFO(First In Not Used First Out) を用いている。この方式は、図 5.14 に示すように、各ページ毎に参照ビット (used bit) を設け、そのページ内への参照がある毎に、対応するビットをセットするものである。ページフォルトが生じると、サーチ・ポインタによって、参照ビットが “0” のページを捜し、それを入れ換え候補ページとする。サーチ・ポインタは、図中矢印のようにポーリングし、参照ビットが “1” ならばそれを “0” にリセットして、つぎのページに移る。

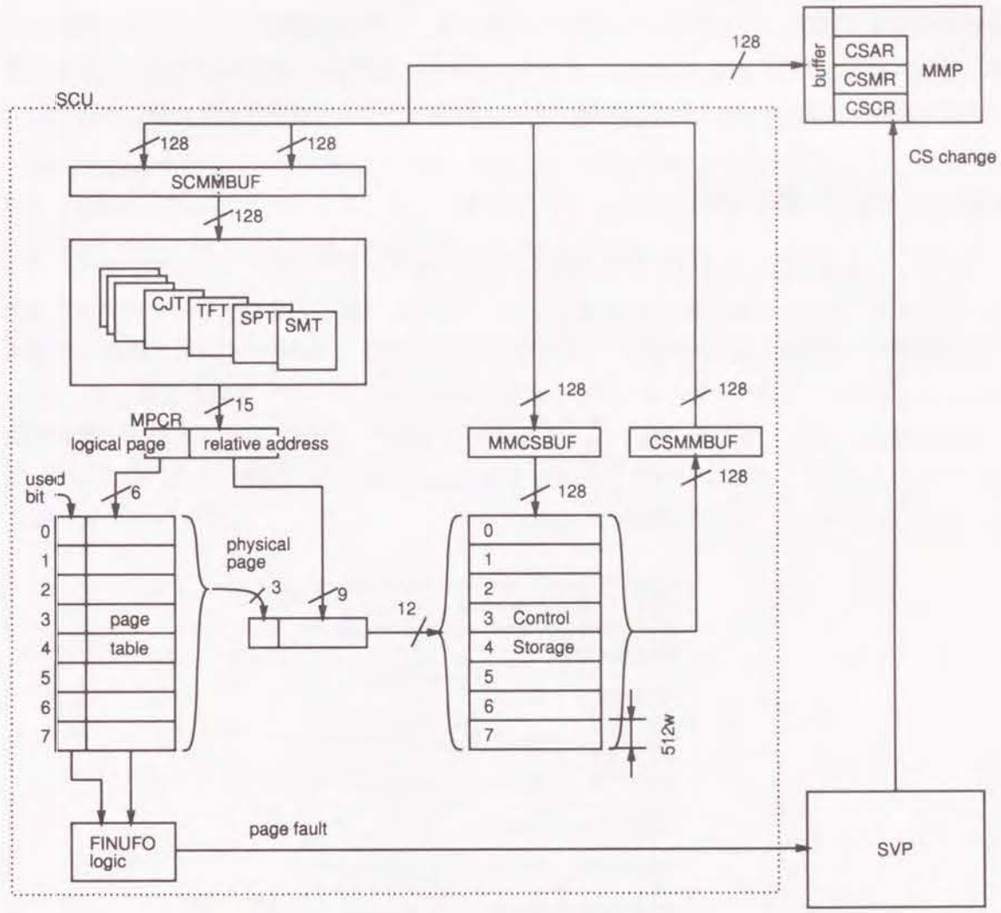


図 5.13: 制御記憶部の構成

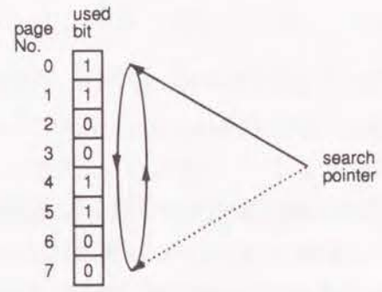


図 5.14: FINNUFO の処理方式

QA-1 の仮想制御記憶方式で用いた LRU(Least Recently Used) 方式は、プログラムのページ参照特性をよく反映しているが、ハードウェア量が多く、実現が困難である。一方、FINUFO は、MULTICS、OS7、VM/370 などで使用されており、能力は LRU に近く、ハードウェア量は FIFO(First In First Out) 程度で実現できる、優れた方式といわれている。[31] [57]

5.6 SVP の要求処理

SVP は、QA-2 システム全体の管理を担っており、マイクロプログラムの制御記憶へのロード、デバック、システム起動等 SVP を通してユーザが指定できる。また、マイクロプログラムの実行状態や、RALU、SCU、MMP の動作状況も全て把握しており、それらの動的な管理も行う。そのために、SVP は常に他の部分と密接に連絡を取り合って、情報を収集している。

SVP が MMP に対して発する要求は、表 5.3に示す通りで、制御記憶の入れ換え、順序制御テーブルの入れ換え、さらに、SVP からの主記憶 Read/Write の 2 種類がある。前者については、5.5で述べたので、ここでは扱わない。

SVPOPR	function
00000000	
00000001	Read
00000010	Read & Increment
00000011	Read & Decrement
00000100	Write
00000101	Write & Increment
00000110	Write & Decrement
00000111	CS Read
00001000	CS Write

表 5.3: SVP の要求

SVP からの Read/Write で使用するレジスタは、SVPAR、SVPDR、SVPRWCR、SVPIDCR の 4 種類であり、これらは RALU からの Read/Write における各レジスタと同様の働きをする。実際には、図 5.4、5.5で示したように、これらのレジスタは、MAR2、MDR2 と同一のバスを用いて処理する。但し、SVP は全主記憶空間に自由にアクセスできることが必要なので、アドレス・チェックは行わない。SVP の Read/Write 要求は、デバック時に全システムを停止して行うので、MMP 内における要求の受け付け優先順位は最も低い。その他のレジスタとしては、SVP からの命令を保持するレジスタ SVPOPR(8 ビット) がある。現在は、表 5.3に示す通り 8 種類の命令しかなく、拡張可能である。

USRUBR、USRLBR は 5.4.2で述べた通り、ユーザー単位に割り当てた、主記憶領域の上限、下限アドレスである。

多重マイクロプログラミングを実現する場合など、ユーザが変わる度に、SCU 内の各種テーブルと USRUBR、USRLBR、そして必要があれば、制御記憶の入れ換えを SVP が管理する。

図 5.15に SVP 用レジスタの構成を示す。SVP は、セレクト信号を出すことによって、これらのレジスタに随時アクセスすることが可能である。SVP からレジスタへの値設定が、MMP の値設定と重なる場合には、MMP のアクセスを優先させる。

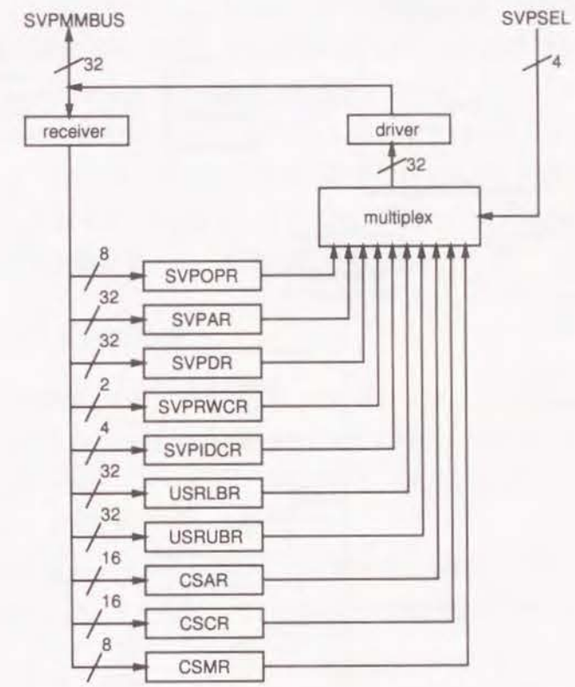


図 5.15: SVP のレジスタ

5.7 外部入出力装置の要求処理

QA-2 の入出力装置として、ディスプレイ装置のように、データの高速転送を必要とするものと、ディスクや各種低速入出力装置等、DMA モードのデータ転送を行うものの 2 種類ある (図 5.16)。

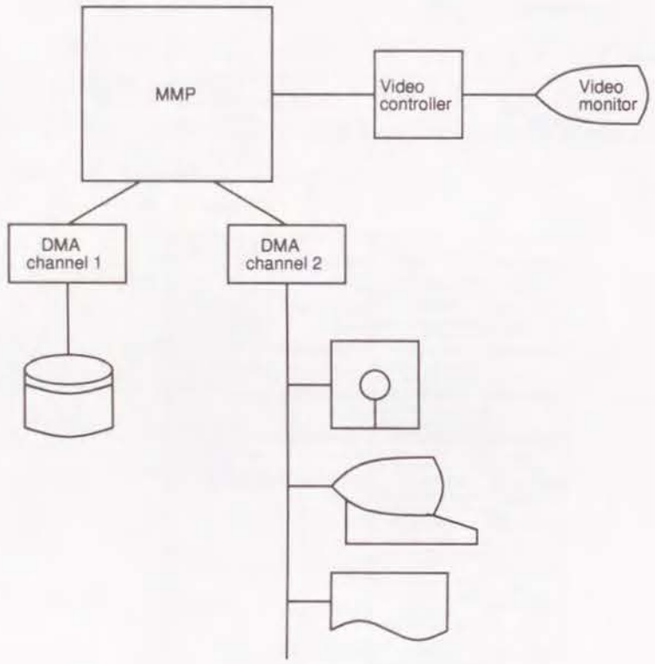


図 5.16: 外部入出力装置との結合方式

5.7.1 ディスプレイ装置

QA-2 の応用分野として、図形処理、画像処理等高速専用計算機のエミュレーションを考えている。すでに、QA-1 上で試作、評価した、実時間動画システム等の経験から、ALU における演算と、画像表示処理とを有機的に結合させるには、画像記憶を主記憶内に設定し、画像データを超高速バースト・モードでディスプレイ装置に転送する方式が、非常に有効であることが分かっている。

そこで、MMP でも主記憶内の任意の領域に画像記憶を設定することができ、指定されたアドレスから、データ転送を開始することができるよう設計している。ディスプレイ装置へは、128 ビットのデータ・バスを利用して、約 40MB/S の高速転送が可能である。動画処理の場合には特に、転送速度の高速化は、表現しうる動画の複雑さ、実時間性の向上につながる。また、高速性を利用すれば、各種ディスプレイ装置の諸機能に効果的に対応できると思われる。MMP は、QA-1 の約 2 倍の高速転送能力を有するので、RALU における演算処理機能の強化、RALU との非同期動作などと合わせて、高速かつ高度な図形・画像処理の実現が期待できる。

転送アドレス (バンク内アドレス)20 ビットのうち、上位 4 ビットは外部スイッチによって設定し、下位 16 ビットを制御装置から、転送の度に受け取る。

5.7.2 DMA モード入出力装置

QA-2 と外部入出力装置とのデータ転送路には 2 通りある。1 つは、レジスタレベルのデータ転送で、RALU 内の IOR(Input/Output Register) に格納したデータを、SVP を介して入出力装置に転送する、プログラムモードの入出力である。他方が DMA チャネルを介して MMP とのデータ転送を行う DMA モードである。

MMP には 2 台の DMA チャネルが接続され、それぞれが独立に要求を発する。MMP-チャネル間の転送は、次のレジスタを利用して、RALU の主記憶参照をサイクルスチールしながら行う。

DMAAR:24 ビット 主記憶内アドレスを指定する。

DMADR:32 ビット Read/Write すべきデータを保持する。

DNARWCR:2 ビット DMADR 内のデータ長を示す。

2 つの DMA チャネルによる要求は、アービタにおける優先順位が異なるだけで、処理方式は全く同じである。実際には、優先順位の高い方のチャネルに、高速ディスク用セレクタ・チャネルを接続し、他方に、各種低速入出力装置 (フロッピーディスク、ラインプリンタ、CRT、etc) 用マルチプレクサ・チャネルを接続する。各装置のデータ幅と MMP のデータ幅との相互変換や、アドレスの管理などをチャネルが行う。

5.8 考察

MMP は、QA-1 上でのマイクロプログラム作成の経験を生かして、QA-1 の主記憶参照における問題点を検討し、それらを改善すると共に、新たに考案した機能を加えた。その結果、MMP は主記憶管理のみならず、QA-2 全体としての処理効率向上に寄与している。そこで、本章では、MMP の特徴や機能を再掲しながら、その効能について考察、検討を加える。

5.8.1 並列処理機能に関する考察

MMP は、RALU における 4 つの ALU の演算能力、並列性、独立性を考慮して、4 組の MAR、MDR を装備している。高度な ALU 機能を充分活用するには、それぞれの ALU が主記憶への独立なアクセスパスをそなえることが必要である。それによって、RALU と主記憶とが密接に、かつ、効果的に結合され、各 ALU は文字通り並列に動作可能となる。以下に、具体的な QA-2 の適用例をいくつかあげて、時に QA-1 と比較しながら、MMP の有効性を考察する。

配列演算

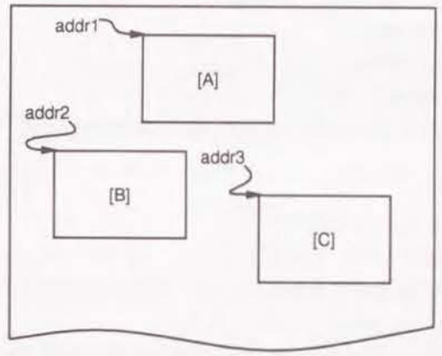
図形/画像処理のように大量の配列データを扱う場合や、高級言語における配列処理など、主記憶内データ間の演算を効率良く行うためには、複数の MAR と複数のアクセス・パスを持つことが不可欠である。

図 5.17(a) のような演算を行う場合、QA-1 のように、主記憶への入口を 1 つしか持たない場合には、図 5.17(b) のように、主記憶アドレスを汎用レジスタに持ち、MAR を主記憶参照の度に変更することになる。この場合、時間的なオーバヘッドもさることながら、マイクロプログラムの論理構造を解読し難くする。

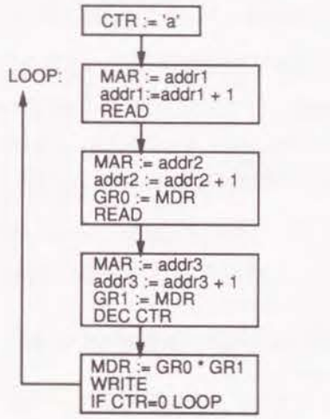
一方、MMP では、A、B、C それぞれの領域を同時に指し示すことができるので、演算処理が人間のアルゴリズム思考過程とうまく合致し、図 5.17(c) のように表すことができる。すなわち、1 ステップループによって配列演算が処理でき、演算式との効率良い対応がとれる。この例では、4 つの MAM フィールドによって、同時に Read/Write が指定できる機能なども大きく作用している。

同様の効果を行列の数値演算処理にもみることができる。図 5.18 は、連立方程式 $PX = Q$ の解を掃き出し法で求める、Algol プログラムである。P、Q がそれぞれ m 行 m 列、m 行 $n-m$ 列の行列で、解 X と、P の逆行列 P^{-1}、P の行列式の値を同時に求めている。(プログラムでは $A = [P|Q]$)

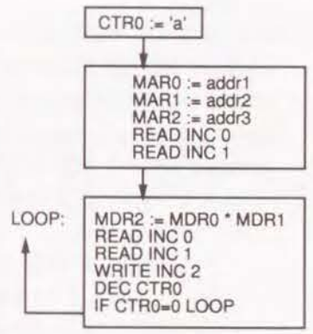
このプログラムで、実際に掃き出しを行っているのは (3)、(4) のループである。特に (4) では、行列の異なる要素が 1 つの演算式に表れている。QA-2 の場合、主記憶への並列アクセスと、ALU 連鎖演算機能を利用すれば、(4) の処理を容易にマイクロプログラムで表現できる。



(a) データ配置



(b) MAR, MDRが1組の場合



(c) QA－2の場合

図 5.17: 配列演算の処理 ($C \leftarrow A \times B$)

```
procedure MATINV (A,M,N,DET);
   value M,N;  integer M,N;  real DET;  array A;
   begin integer I,J,K,PIVR,INDEXK;  real W,PIV;
   integer array INDEX(1:M);
   DET:=1.0;  comment INITIALIZATION
   for K:=1 step 1 until M do
      begin
         comment SEARCH FOR PIVOT ELEMENT;
         PIV:=0.0;  PIVP:=0;
         for I:=K step 1 until M do ..................................(1)
            begin W:=ABS(A[I,J]);
               if PIV > W then begin PIV:=W;  PIVR:=I end
            end;
         INDEX[K]:=PIVR;
         if PIVR=0 then go to END;
         if PIVR=K then
            comment INTERCHANGE ROWS TO PUT PIVOT ELEMENT ON DIAGONAL;
            begin for J:=1 step 1 until N do ..........................(2)
                     begin W:=A[PIVR,J]; A[PIVR,J]:=A[K,J]; A[K,J]:=W
                     end;
                  DET:=-DET
            end;
      PIV:=A[K,K]; DET:=DET * PIV;
      comment DEVIDE PIVOT ROW BY PIVOT ELEMENT;
      A[K,K]:=1.0;
      for J:=1 step 1 until N do A[K,J]:=A[K,J] / PIV; .............(3)
      comment REDUCE NON-PIVOT ROWS;
         for I:=1 step 1 until M do
            if I=K then
               begin W:=A[I,K]; A[I,K]:=0;
                  for J:=1 step 1 until N do .......................(4)
                     A[I,J]:=A[I,J] - A[K,J] * W
               end
   end;
   comment INTERCHANGE COLUMNS;
      for K:=M - 1 step -1 until 1 do
         begin INDEXK:=INDEX[K];
            if INDEXK=K then
               for I:=1 step 1 until M do ...........................(5)
                  begin W:=A[I,INDEXK]; A[I,INDEXK]:=A[I,K];
                        A[I,K]:=W
                  end
         end;
   END:
end;
```

図 5.18: 行列演算プログラム

また、(2) は枢軸 (pivot) を対角線上に移動させるための列交換、(5) は逆行列生成のための行交換処理である。行列演算ではこのような入れ換え処理が、行列の転置をはじめ頻繁に現れる。MMP では、主記憶への複数のアクセスパスを利用して、一時待避領域を用いることなく入れ換えを実現できる。このように、本質的に並列処理が可能なデータ構造を扱う場合、MMP の機能はアルゴリズムに対して、かなり効率良く対応することができる。

一方、主記憶へのアクセスパスを 1 つしか持たない場合には、このような処理にかなりのオーバヘッドを要するのは、配列演算の場合と同様である。

フェッチ・ルーチン

ファームウェアを利用した高級言語処理では、高級言語を一旦中間言語に変換して、中間言語を解釈、実行 (インタプリト) する方式が一般的である。主記憶に格納した中間言語のフェッチには、命令フェッチ、オペランド・フェッチの 2 段階があり、それぞれの処理で独立の MAR を必要とする。つまり、1 つはプログラム・カウンタ専用に使い、他方はオペランド・フェッチの際に変更しながら用いる。同様のことが、マシン・エミュレーションにおける、ターゲット計算機の命令フェッチの際に要求される。MMP では、1 つの MAR をプログラム・カウンタとして専用に用いても、残る 3 つの MAR をオペランド用に使用できるので、QA-1 で指摘されたような、プログラム・カウンタの待避、復元等のオーバヘッドが生じない。

また、RALU との非同期動作を利用して、オペランドの処理と次の命令フェッチを並行して行うことが可能なので、フェッチ・ルーチンによるオーバヘッドは殆ど無視できる。

演算に対する複数オペランドへのアクセスは、本質的に並列処理可能なものが多く、また、演算結果の格納と、次の演算オペランドへのアクセスは、同時指定できる場合が多い。このように、複数 MAR、MDR を利用した主記憶への並列アクセス機能は、他の多くの応用でも充分威力を発揮するものと考えられる。

5.8.2 データ検索機能に関する考察

MMP のデータ検索機能は、次の 4 つの特徴を備えることによって、非常に使い易く、かつ、強力になっている。

1. 検索データに対してユーザが自由にマスクをかけられる。
2. 検索アドレスの増分をユーザが自由に指定できる。
3. データ発見をフラグで、異常終了を割り込みで知ることができる。
4. 検索中も RALU の主記憶 Read/Write が可能である。

データ検索機能の用途としては、その名の通り、領域内のデータ検索や、連想記憶としての使用などが考えられる。

まず、検索機能をハードウェア化したことによる効果を、プログラムを通して考察する。

図 5.19のように、あるテーブルの 1 エントリ (2 バイト) の中央 1 バイトが "C0" であるデータを探す場合を考える。

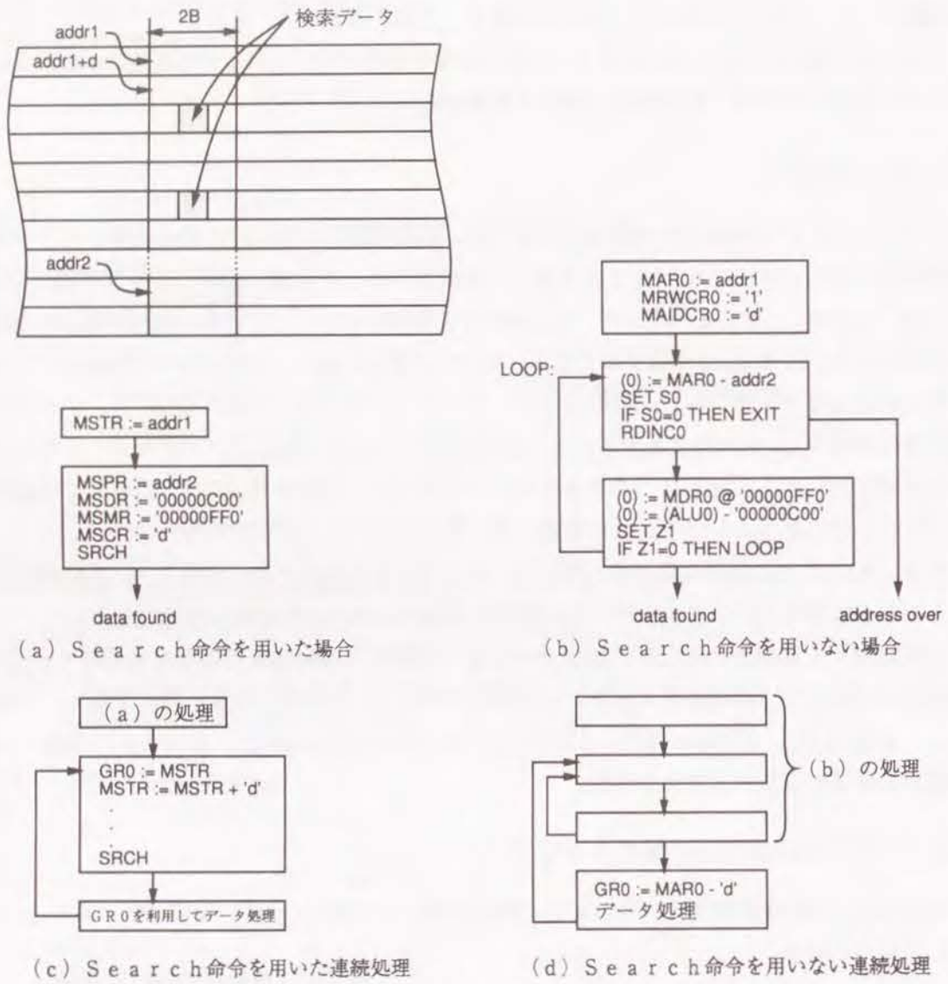


図 5.19: テーブルの検索処理

マイクロプログラムで全て処理するならば、(b) のように、アドレスチェック、データ・チェックのループを回ることになる。((b) では ALU 連鎖演算機能を利用している。) 一方、Search 命令を用いれば、(a) のように、初期値設定に必要な 2 ステップだけで良い。次に、同じ値をもつデータを次々と検索しながら、そのデータを処理する場合を考える。この時、ハードウェア化の利点が歴然とする。即ち、プログラムで処理すると、(d) のように、データ処理後、次

のデータ発見まで主記憶参照を繰り返すため、処理時間の大半を主記憶へのアクセスに費やしてしまう。一方、Search 命令を用いると、(c) のように、データの処理と検索を、完全に並行して行えるので、検索によるオーバヘッドを最小限に留めることができる。

このように、検索と並行処理できる仕事がある時 (プログラムをそのように組むことは比較的容易)、ハードウェア化が生きてくると言えよう。

連想記憶は、今日、高機能、大容量のものが試作、製作されており、その応用としては、リスト処理、画像処理、データベース処理などが考えられる。[58] MMP のデータ検索機能は、4 バイト単位の逐次処理だから、データベース処理のように膨大なデータの検索には利用できないが、比較的小容量のデータに対しては、ALU 演算と組み合わせて、仮想的な連想記憶を構成することはできる。最近の連想記憶は、大小関係だけでなく、最大値、最小値、2 番目に大きい (小さい) 値など、多くの判断機能を備えている。そこで、必要なデータの候補を検索機能によって求め、その後複雑な判定を ALU で行い、最終的に判断するという方式を用いる。

例えば、高級言語処理において、データへのアクセスを、ディスクリプタと連想記憶を用いて実現するシステムがある。[29] この時、ディスクリプタのデータ・タイプなどをキーワードに検索を行い、該当するデータに対して ALU でチェックを行う、という方法が考えられる。この場合にも、ALU でのチェックと検索は並行動作ができる。このように、並行動作を充分活用できる応用に対して、Search は効果的である。

5.8.3 ブロック転送機能に関する考察

Move は、マイクロプログラムで処理すれば、主記憶への Read、Write 操作の繰り返しになる機能部分を、ハードウェアで実現した機能である。その用途としては、配列間の代入や、文字列処理、領域移動などがあげられる。

配列データを主記憶内に保持している場合、それに対する代入操作は、単なるデータ転送であり、システムはその間に、次の演算処理を行うことができる。この時、ブロック転送機能を利用すれば、転送と並行して、ALU 演算や、主記憶アクセスが可能なので、システムの処理効率を上げることができる。

文字列の処理においては、削除、挿入、連結等の作業が多い。長大な文字列を対象とする場合、記憶域の節約のため、削除した部分を詰め合わせる等、データの転送が頻繁に生じる。この場合も、ブロック転送機能を利用することができる。

さらに、出力データを、プログラムの最終段階で出力領域に転送する場合、あるいは、ユーザのプログラムやデータ全てを主記憶内で移動する場合などにも、ブロック転送機能を用いて、他の処理との並行動作が行える。

このように、ブロック転送機能は、大量のデータ転送を行って、その転送終了を待つ必要のない場合に、効果的に利用することができる。

Chapter 6

システム制御部の構成

6.1 まえがき

汎用エミュレーション [20] を指向したユニバーサル・ホスト計算機 QA-2 は、低レベル並列処理機能 (算術論理演算・レジスタ転送レベルでの並列処理) をマイクロプログラム方式で制御するというシステム構成法のもとに開発され [26] [53]、図形・画像・信号処理などのリアル・タイム処理や高級言語処理などを始めとする多様な応用に利用されている [55] [44] [72] [24]。

QA-2 では、図 6.1に示すようにレジスタ・ALU 部 (RALU) と順序制御部 (SCU) から成る CPU と、主記憶管理プロセッサ (MMP)、システム管理プロセッサ (SVP) の 3 つの機能部分を独立に構成している。CPU 部では、4 台の均質な ALU が水平型マイクロ命令 (1 ワード=256 ビット) の相異なるフィールドによって独立に制御され、6K バイトのレジスタを共有して並列演算を実行できる。また MMP は 16 ウェイ・インタリーブ構成によって 4 つの主記憶アクセスを同時に行うことができ、さらにデータ探索など高機能操作を CPU と独立に実行できる構造になっている。

CPU を構成する RALU と SCU および MMP のハードウェア構成については、既に他の章で述べたので、本章ではシステム管理プロセッサ (SVP) のハードウェア構成、特に、機械命令レベルの (マクロ) アーキテクチャとそれをサポートするマイクロ命令レベルの (マイクロ) アーキテクチャについて詳述する。

6.2 SVP のマイクロ・アーキテクチャ

6.2.1 マイクロ・アーキテクチャの設計指針

SVP の基本仕様は、

- マイクロプログラム・デバッガやエバリュエータおよび実行モニタなど、QA-2 のマイクロプログラミング支援を行う強力な計算機となり得ること、

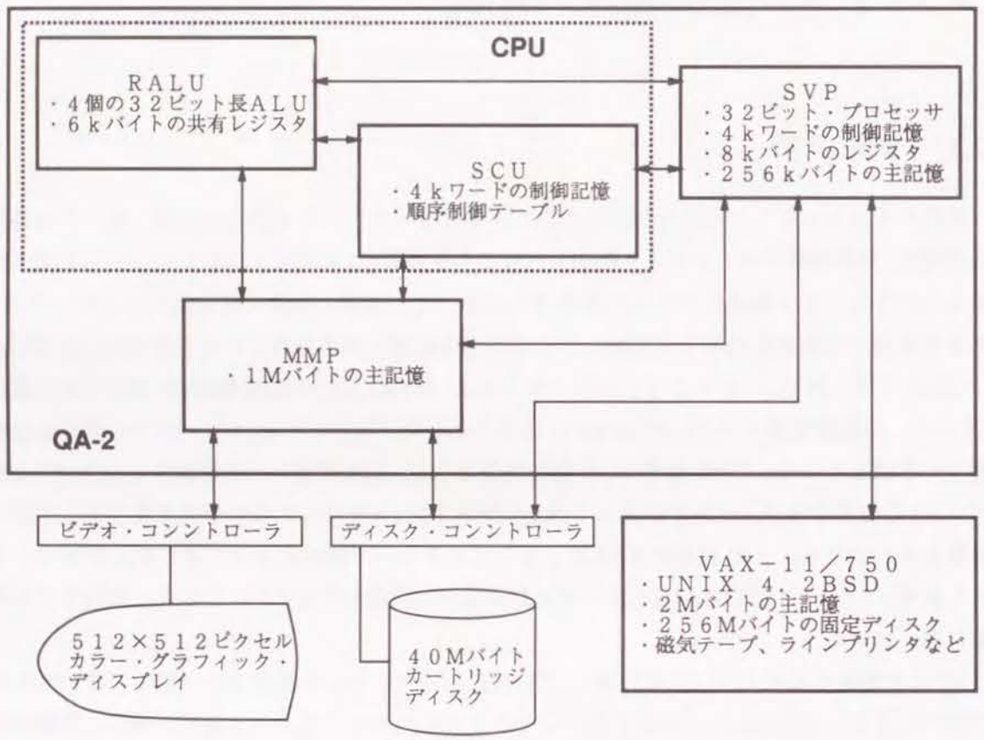


図 6.1: QA-2 のシステム構成

- ファイル装置や研究室内ネットワークも含めた外部入出力装置の管理などの通常の OS 機能を高速に実現できること、
- QA-2 上で種々の仮想計算機のエミュレーションを並行して行うために制御記憶 (CS) や制御テーブルのダイナミックな切り換えを行うマルチ・マイクロプログラミング環境を実現できること、

などである。これらの機能を実現するシステム・プログラムは高速性が要求され、また大規模なものとなるので、プログラムの生産性や処理効率を向上し得るよう、

- システム 記述専用高級言語の処理に適した機械命令の装備、
- マルチプロセスの効率の良い実行環境の提供、
- ソフトウェアのモジュール化の支援など、

SVP 機械命令の機能をシステム管理向きに強化することが必要である。
これらの高機能機械命令をサポートする SVP のマイクロ・アーキテクチャは、

- 極度なハードウェア化は抑えて、他の高級言語処理への対応やマイクロプログラムの生産性向上も容易に達成できるよう柔軟性に富んだクリーンな構成方式とすること、
- システム管理に重要な役割を果たす主記憶管理機構や主記憶アクセスの高速化を図ること、
- 可変長機械命令のフェッチ・デコードのオーバーヘッドを極力減らすこと、

などを念頭に設計されている。

6.2.2 マイクロ命令形式とバス構成

SVP のマイクロ命令は図 6.2に示すように 1 語 114 ビットからなる簡潔な水平型となっている。マイクロ命令の各フィールドは、図 6.3に示すハードウェア・モジュールを制御している。タイミング制御部以外のモジュールは、L-BUS、R-BUS、D-BUS の 3 種のバスにより結合されている。各バスの幅は 32 ビットであり、レジスタ部 (WR、IRF など)、及び各モジュールの特殊レジスタの内容が、L-BUS、R-BUS を通って演算部へ転送され、演算結果が D-BUS を通ってレジスタ部、及び特殊レジスタに転送される。

6.2.3 演算部の構成

演算部では図 6.2のマイクロ命令の OP-フィールドで指定する演算の種類と、LG-フィールドで指定する演算長とに従って、入力データに演算を施す。演算長は、8、16、及び 32 ビットであり、LG-フィールドの最上位ビットが “0” の場合は、LG-フィールドの下位 2 ビットにより

図 6.2: SVP のマイクロ命令

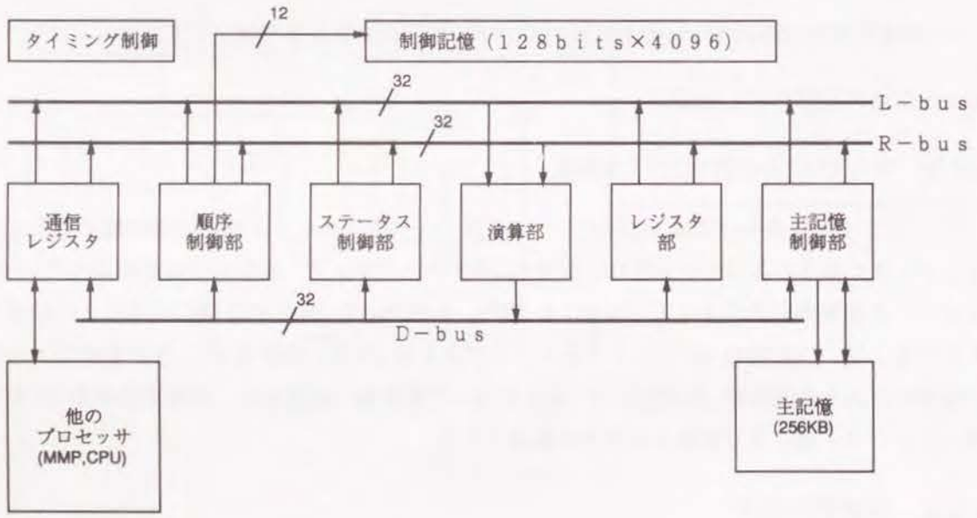


図 6.3: SVP の概要

指定される。最上位ビットが“1”の場合は、特殊レジスタ LGR(operation LenGth Register) の内容により指定される。

演算結果はマイクロ命令の MK-フィールドで指定されるマスク値との論理積を施された後、D-BUS に出力される。

6.2.4 レジスタ部の構成

レジスタ部には、2 ポートの汎用レジスタ (WR: Work Register) と、特殊レジスタ (SR: Special Register) および間接レジスタ・ファイル (IRF: Indirect Register File) が装備されている。WR と SR は、マイクロ命令の L-フィールド、R-フィールドの最上位ビットが“1”の時に下位 8 ビットよって直接アクセスされ、その内容がそれぞれ L-BUS、R-BUS に出力される (最上位ビットが“0”の場合には 8 ビットの定数値となる)。D-フィールドでは D-BUS 上のデータを格納するレジスタを指定する。

WR はバイト単位でアクセスされ、容量は 64B である。ハードウェア的には 4 バンク構成 (8 ビット×4) となっているため、演算長に合わせて、連続したバイト・アドレスへのアクセスが可能である。

一方、IRF は、SR の一つである IAR(Indirect Address Register) の内容をアドレスとして、間接アクセスのみが可能なレジスタである。SVP は 1K バイト (256 エントリ×32 ビット) の IRF を 8 個装備している。IAR には自動増減機能が装備されており、IRF は高速なスタックやキューなどを容易に構成できる。

6.2.5 ステータス制御部の構成

システム内のステータス・レジスタは 98 ビット長である。マイクロ命令の CF0、及び CF1 の 2 つのフィールドにより陽にステータス・レジスタのアドレスと操作を指定することによって、1 マイクロ命令で 2 つのステータスの設定が可能である。

マイクロ命令の TS0 および TS1 の 2 つのフィールドにより、1 マイクロ命令で同時に 2 つのステータス (論理変数 (A1,A0)) を読み出すことができる。TF-フィールドは 4 ビット用意されており、TS0 と TS1-フィールドで指示された 2 つの論理変数 (A1,A0) に対する任意の論理関数 $f(A1, A0)$ (恒等的に 1,0 の場合を含む) を指定できるため、2 方向分岐などで複雑な条件設定ができる。

6.2.6 順序制御部の構成

分岐形式はマイクロ命令の BR-フィールドで指定され、次のようなものが用意されている。

単純分岐 マイクロ命令の AD-フィールドの値をマイクロプログラム・カウンタに設定する。

形式は

if $f(A1, A0)$ then GOTO AD else NEXT

サブルーチン・コール 復帰番地をアドレス・スタック (深さ 16) にプッシュし、AD-フィールドの値を MPC に設定する。

形式は

if $f(A1, A0)$ then CALL AD else NEXT

4 方向分岐 分岐条件として選択された 2 つのステータスの値を 2 ビットの整数とみなし、AD-フィールドの値との和を MPC に設定する。

形式は

```
case (A1, A0) of /0/ GOTO AD,
                 /1/ GOTO AD+1,
                 /2/ GOTO AD+2,
                 /3/ GOTO AD+3
```

テーブルを用いた間接分岐 特殊レジスタである JTPR (Jump Table Pointer Register) の内容で JTB (Jump TaBle : 4K エントリ) を索引し、その値を MPC に設定する。一般的には、機械命令コードを JTPR に設定し、各実行マイクロ・ルーチンへの分岐を行う。

IJR(Indirect Jump Register) を用いた多方向分岐 IJR の内容と AD-フィールドの値との和を MPC に設定する。

形式は

```
if f(A1, A0) then case (IJR) of
                 /0/   GOTO AD,
                 /1/   GOTO AD+1,
                       ⋮
                 /n-1/ GOTO AD+n-1,
                 else  NEXT
```

サブルーチンからの復帰 アドレス・スタックをポップ・アップしてその値を MPC に設定する。

形式は

if $f(A1, A0)$ then RETURN else NEXT

6.2.7 主記憶制御部の構成

主記憶制御部は、次章に示す機械命令コードやオペランドの高速フェッチを可能とするよう特殊なハードウェア設計がなされている。

高速アドレス生成機構

SVP には 8 個の MAR(Memory Address Register) が装備されており、この中から任意の 2 個をマイクロ命令の MS0、MS1-フィールドにより選択する。主記憶アドレスはこれら 2 つの MAR をハードウェアで直接加算して得られる。MS0-フィールドにより選択された MAR に対しては、自動増減を行うことができる。その際、加算または減算する値は、MLG-フィールドで指定されるアクセス長に等しい。アクセス長は、8、16、24、および 32 ビットのいずれかであり、MLGR(Memory access LenGth Register) による間接指定も可能である。

また、SVP には領域チェックのために、4 組の UBR と LBR(Upper / Lower Boundary Register) が装備されている。これらのレジスタで、上限、下限を指定することにより、4 つの領域に対するチェックを高速に行うことができる (MBM-フィールドで指定)。

高速データ・アクセス機構

主記憶のアクセス動作は、MOP-フィールドにより次のように指定される。

読み出し 読み出したデータを MBR(Memory Buffer Register) に格納する。読み出し動作の間、他のハードウェアは動作しない (CPU フリーズ)。

連続読み出しの開始 指定されたアドレスで読み出したデータを MBR に格納した後、そのアドレスにアクセス長を加えた値をアドレスとして再度読み出し SRB (Sequential Read Buffer) に格納する。CPU フリーズは 1 回目の読み出し時にのみ起こり、2 回目は、主記憶制御と他の演算・制御が並列に行われる。

連続読み出し 直前に実行した連続読み出しで格納された SRB の内容を MBR に格納するとともに、指定されたアドレスにアクセス長を加えた値をアドレスとして読み出し SRB に格納する。CPU フリーズは起こらない。

書き込み MBR の内容を書き込む。CPU フリーズは起こらない。

6.3 SVP のマクロ・アーキテクチャ

6.3.1 システム記述用高級言語向き機械命令セットの設計指針

システム記述用高級言語のための機械命令を設計するにあたっては、効率の良いシステム・プログラムの実現と、多重プログラミング環境における信頼性の向上を目的として、以下に述

べるような方針をとった。

機械命令の機能の強化

機械命令の機能を強化することによって、機械命令と高級言語との間のセマンティック・ギャップを狭め、多様なアドレシング・モード (データ・アクセス用 20 種、分岐用 6 種) を装備し配列やレコードに対するアクセスにおけるオーバヘッドの軽減を図った。

ソフトウェアのモジュール化支援

大規模なプログラムを作成するにあたっては、プログラムをモジュール化することが必要不可欠である。SVP ではモジュール化の概念を機械命令レベルで積極的にサポートしている。SVP の機械語のプログラムは、複数個のモジュールからなる。各モジュールは一連のまとまったデータおよびそのデータを操作するサブルーチン群によって構成される。モジュール間のデータの受け渡しはサブルーチン間のパラメータの受け渡しとしてのみ可能である。この受け渡しを効率良く行うために値呼び、番地呼びの形式によるパラメータを用いたサブルーチン・コールを機械命令レベルで実現している。

図 6.4に主記憶の構成を示す。主記憶を、 (a) 命令語領域、 (b) 個々のモジュールに固有のモジュール・データ領域 、および (c) パラメータの受け渡しやサブルーチンの局所変数を保持するスタック領域、の 3 つに分割した。さらに各領域外への不当なデータ・アクセスの防止やプログラムの暴走の回避を図っている。

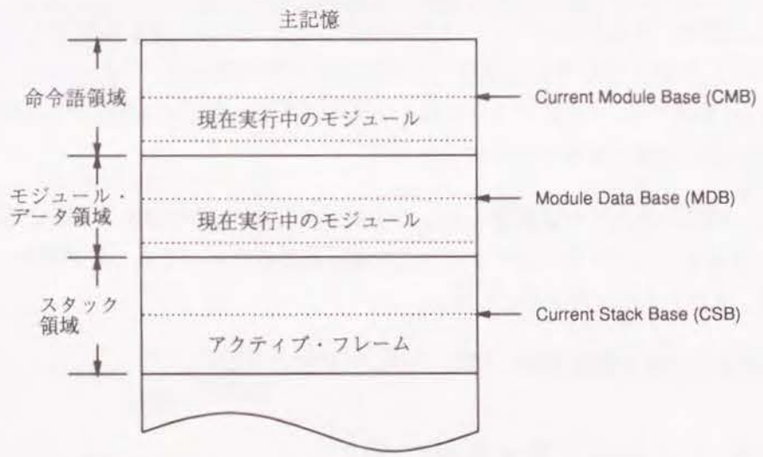


図 6.4: 主記憶の構成

また、処理の高速化のために、実行中のプロセスのスタック領域の上の部分を SVP の間接

レジスタ・ファイル (IRF) にコピーしている。

マルチプロセスの実行環境の整備

マルチプロセスの実行を効率良く行うためには、 (a) コンテクスト・スイッチング時におけるオーバヘッドを軽減するとともに、 (b) 臨界領域 (複数個のプロセスによって共有されるデータ) に対する操作を効率良く行わなければならない。

(a) に対しては、プロセスの生成/消滅を行う fork 命令/join 命令やプロセス間の同期を行うセマフォアの P/V 操作に対応する wait 命令/signal 命令を機械命令として装備した。 (b) に対しては、Hoare の提唱したモニタ [18] の概念を機械語に導入し、モジュールの一種として、モニタ・モジュールという特殊なモジュールを用意した。各モニタ・モジュールには複数個のプロセスに共有されるデータ、そのデータに対して施される操作、およびモニタ・ロックという名のセマフォア変数を割り当てている。このモニタ・ロックに対して前述した wait 命令や signal 命令を施すことにより、臨界領域に対するプロセスの相互排斥が実現できる。

6.3.2 機械命令の形式

図 6.5に SVP の機械命令の形式を示す。機械命令の固定長部分は命令コード (OP) とオペランド長の指定フィールド (L) 及び分岐条件指定フィールド (MOD) からなる。

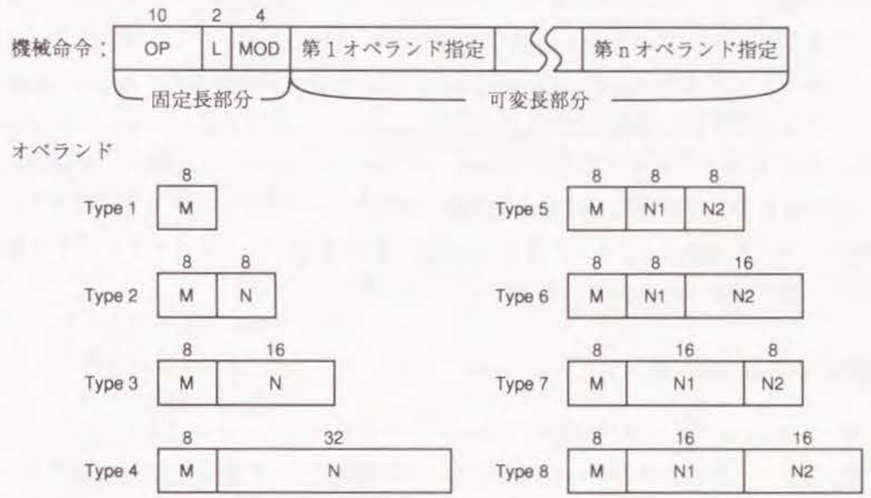


図 6.5: 機械命令の形式

機械命令の可変長部分はいくつかのオペランド指定部からなる。オペランド指定部の個数はほとんどの命令では定まっているが、call 命令や fork 命令ではパラメータの数によって可変で

ある。オペランド指定部はアドレッシング・モードを指定するフィールド (M) とアドレス指定フィールド (N) から成っており、図 6.5 に示すように 8 種類の形式をとる。

6.3.3 アドレッシング・モード

データ・アクセス用アドレッシング・モード

このモードは図 6.4の主記憶の構成を反映して、以下のように 3 種に大別される (表 6.1参照)。

定数アクセス 1 バイト、2 バイト、または 4 バイトの定数が指定できる (表 6.1のモード番号 0 ～2)。

モジュール・データ・アクセス 実行中のモジュールのモジュール・データのアドレスを直接、または間接に指定する。間接指定の場合はポインタ・データの自動増減 (表 6.1のモード 5、6)、2 個のポインタの和による指定 (モード 9、10)、2 重間接指定 (モード 7、8) など豊富なアクセス方法がある。

スタック・アクセス スタック領域に割り当てられるデータとしては、サブルーチンへ渡されるパラメータ、サブルーチンで動的に割り当てられる変数 (ローカル・データ) の 2 種類がある。各サブルーチンには最大 1K バイトの SF(Stack Frame) が割り当てられる。 サブルーチンへのパラメータの渡し方としては、値呼びと番地呼びの 2 種類がある。番地呼びの場合、表 6.1のモード番号 12～19 に示すように、そのパラメータが同じモジュールのモジュール・データ (mod) か、他のモジュールのモジュール・データ (ext) か、あるいはスタック上のデータ (stk) かによってアクセス方法が異なる。mod,ext,stk のいずれのアクセス方法を採るかについての情報はパラメータ自身にタグとして付加されている。 ローカル・データへのアクセスは、サブルーチンが呼ばれた時点で自動的に割り当てられるもの (C 言語における自動変数に対応する) に対するアクセスに相当する。アクセス・モードは表 6.1のモード番号 11～19 に対応するが、パラメータへのアクセスと異なり、他のモジュール内データへのアクセスは行われない。

分岐用アドレッシング・モード

このモードにはモジュール内分岐とモジュール外分岐の 2 つがある (表 6.2参照)。モジュール内分岐/モジュール外分岐いずれの場合にも、直接指定 (モード番号 0/モード番号 3)、スタック・データによる間接指定 (モード番号 1/モード番号 4)、モジュール・データによる間接指定 (モード番号 2 / モード番号 5) がある。モジュール外分岐の場合 (モード番号 3～5)、モジュール番号がロード時にしか定まらないため、モジュール・データ内のモジュール番号辞書を参照する必要がある。モジュール番号から実際のモジュールのアドレスを決定する方法については次節で述べる。

モード番号	モード	注
0	N	i
1	N	ii
2	N	iii
3	mod(N)	
4	mod(mod(N))	iv
5	mod(mod(N))+	iv , v
6	mod(mod(N))−	iv , vi
7	mod(mod(mod(N)))	vii
8	mod(mod(mod(N)))+	v , vii
9	mod(mod(N_1)+N_2)	viii
10	mod(mod(N_1)+mod(N_2))	viii
11	stk(CBS+N)	
12	mod/ext/stk(stk(CBS+N))	iv
13	mod/ext/stk(stk(CBS+N))+	iv , v
14	mod/ext/stk(stk(CBS+N))−	iv , vi
15	mod/ext/stk(stk(CBS+N_1)+N_2)	viii
16	mod/ext/stk(stk(CBS+N_1)+stk(CBS+N_2))	viii
17	mod/ext/stk(stk(CBS+N_1)+mod(CBS+N_2))	viii
18	mod/ext/stk(mod/ext/stk(stk(CBS+N)))	vii
19	mod/ext/stk(mod/ext/stk(stk(CBS+N)))+	v , vii

注) mod : モジュール・データ
stk : スタック
ext : 他のモジュールのモジュール・データ
CBS : Current Stack Base
i 8–bit 定数
ii 16–bit 定数
iii 32–bit 定数
iv 間接アクセス
v ポインタの自動増加
vi ポインタの自動減少
vii 二重間接アクセス
viii 二つのポインタの和によるアクセス

表 6.1: データ・アクセス用アドレッシング・モード

モード番号	モード	注
0	CMB+N	i
1	CMB+stk(CSB+N)	i
2	CMB+mod(N)	i
3	MBT(mod(N_1))+N_2	ii
4	MBT(MOD(N_1))+stk(CBS+N_2)	ii
5	MBT(mod(N_1))+mod(N_2)	ii

注) mod : モジュール・データ
stk : スタック
MBT : 各モジュールの命令語領域の先頭番地を保持
CMB : 現在アクティブなモジュールの
命令語領域の先頭番地
i モジュール内分岐
ii モジュール外分岐

表 6.2: 分岐用アドレッシング・モード

6.3.4 機械命令実行のための資源割り当て

本節では機械命令の処理に 6.2で述べた SVP のマイクロ・アーキテクチャがどのように活用されているかを記憶領域の各部に対応させて説明する。

命令語領域

図 6.6に示すように、命令語領域はモジュールごとに分割されており、各モジュールの先頭アドレスは MBT(Module Base Table、IRF0 に割り付け) に格納されている。実行中のモジュールの先頭アドレスは CMB(Current Module Base 、MAR2 に割り付け) に格納され、実行中の命令の CMB からの相対アドレスは PC(Program Counter 、MAR0 に割り付け) に格納されている。命令語領域の上、下限は UBR0、LBR0 に設定されている。表 6.2のモジュール外分岐では割り当てられたモジュール番号を保持するモジュール番号辞書を参照して実際のモジュール番号を得る、また MBT を用いて主記憶上の番地を得る。

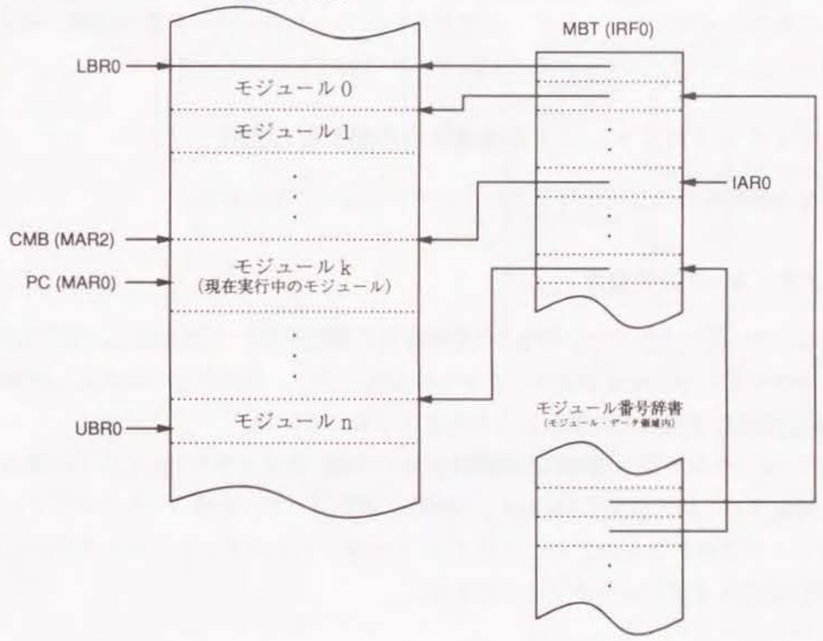


図 6.6: 命令語領域の概要

モジュール・データ領域

モジュール・データ領域はモジュールごとに分割されており、1つのモジュールに対して最大64KBまで設けることができる。各モジュール・データの先頭アドレスはMDBT(Module Data Base Table、IRF2に割り付け)に格納されている。実行中のモジュールに対応するモジュール・データ領域の先頭アドレスはMDB(Module Data Base 、MAR4に割り付け)に格納される。モジュール・データ領域の上、下限はUBR1、LBR1に設定される。

スタック領域

スタック領域は各プロセスに対して割り付けられる(図6.7参照)。各プロセスに対応するスタック領域の先頭アドレスはSBT(Stack Base Table、IRF1に割り付け)に格納されている。実行中のプロセスに対応するスタック領域の先頭アドレスはSB(Stack Base、MAR3に割り付け)に格納されている。実行中のサブルーチンのSFの先頭アドレスはCSB(Current Stack Base、MAR5に割り付け)に格納されている。また、スタック領域の上、下限はUBR2、LBR2に設定されている。

サブルーチンが呼ばれた時点では、図6.7のアクティブ・フレームに示すようにSFには直前に起動されたサブルーチンのCSB、復帰アドレス、およびパラメータが格納されている。サブルーチンの最初の命令でサブルーチンで使用されるローカル・データ用の領域が確保される。6.3.3でも述べたように、スタックの上部のデータはIRF3にコピーされている。

6.3.5 マイクロプログラムによる機械命令の解釈実行過程

機械命令の解釈実行は以下のマイクロルーチンによってなされる。

命令フェッチ・ルーチンの処理

命令フェッチ・ルーチンでは、図6.5の機械命令の固定長部分の読み出し、割り込みの検出、および命令コードに対応する実行ルーチンへの分岐を行う。機械命令の読み出しは連続読み出し機能(6.2.7参照)を用いて先読みによる高速化を図っている。

フェッチ・ルーチンで読み出された固定長部分のうち、命令コード(OP)はJTPR(6.2.6参照)に、長さ指定フィールド(L)はLGR(6.2.3参照)に格納される。命令コードに対応する実行ルーチンのマイクロプログラムのアドレスはJTBに格納されており、テーブルを用いた間接分岐(6.2.6参照)により各実行ルーチンに分岐する。

オペランド参照ルーチン

オペランド参照ルーチンでの処理は、連続読み出しにより、アドレッシング・モード(M)を読み出してIJRに格納する; IJRによる多方向分岐(6.2.6参照)を用いてアドレッシング・モー

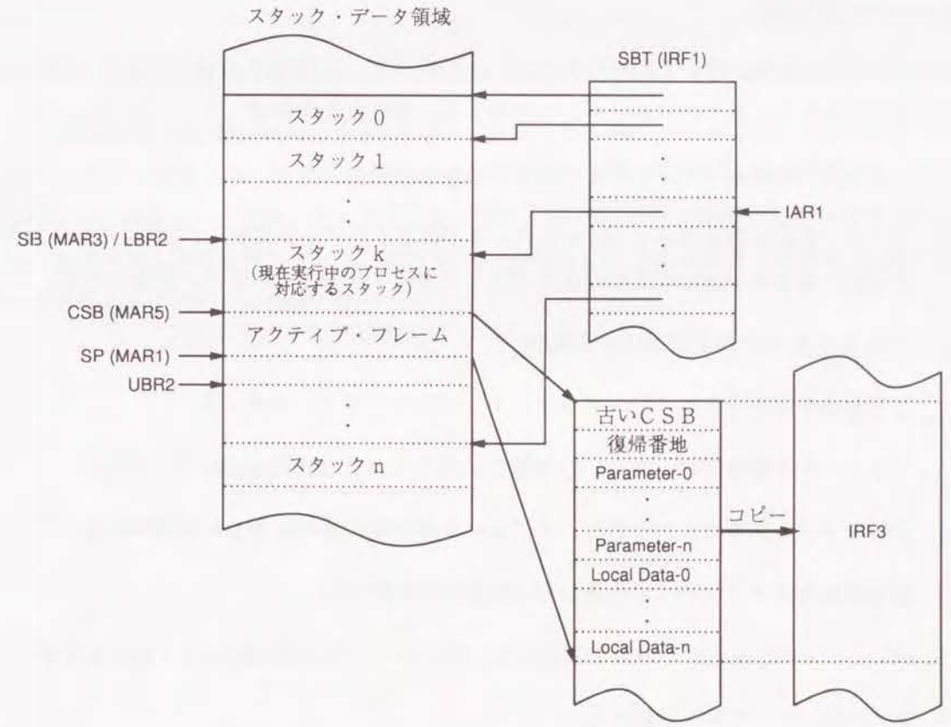


図 6.7: スタック・データ領域の概要

ドに対応するルーチンに分岐する; 連続読み出しにより、アドレス指定フィールド (N) を読み出す; アドレス計算を行い、オペランドへのアクセスを行う; からなる。 スタック上のデータへアクセスする場合、現在実行中のサブルーチンの SF のデータを読み出すときには IRF3 上のデータを用いる。逆に実行中のサブルーチンの SF にデータを書き込む時は IRF3 および主記憶の両方にデータを書き込む。

命令フェッチ・ルーチンとオペランド参照ルーチンにおいては 2 個の MAR の和による主記憶アクセス機能を多用している。

実行ルーチンの処理

例えば CALL 命令に対する処理は次のように行われる。 分岐条件を調べ、条件が偽ならば PC の値を調整してフェッチ・ルーチンへ分岐する。条件が真ならば

1. オペランド参照ルーチンを呼んでパラメータの個数を得る;
2. パラメータの個数を用いて IRF3 にデータが納まるかどうか判断し、IRF3 にデータが納まらない場合は IRF3 の後半の 512 バイトのデータを IRF3 の前半に移動させる;
3. CSB をスタックおよび IRF3 に積む;
4. 復帰番地を保存するためにスタック・トップのアドレスを退避する;
5. パラメータを番地呼び、あるいは値呼びに応じてスタック及び IRF3 に積む;
6. 分岐先アドレスをフェッチする。モジュール外分岐の場合は MDB を更新する;
7. 復帰番地をスタックの 4で退避していた番地に格納する;
8. PC にモジュール内アドレスを設定する。モジュール外分岐の場合は CMB を更新する;
9. フェッチ・ルーチンへ分岐する。

6.3.6 マクロ・アーキテクチャの考察

SVP ファイル・システムのプログラムの一部分を Pascal-like な高級言語で記述した例では、Pascal-like の高級言語 8 ステップが SVP の機械命令 15 ステップに相当していた。また、比較のために IBM370 アーキテクチャの機械語で同じプログラムを記述したところ 51 ステップを要した。これらは、

1. 機械命令レベル・アーキテクチャにレジスタという概念を導入せずにすべて主記憶間での演算としたため、不必要なデータ転送が省けたこと、
2. 複合分岐 (演算を行って、その結果によって分岐する) 命令を活用できたこと、

3. 豊富なアドレッシング・モードを活用すると、配列やレコードに対する処理が容易となったこと、
4. サブルーチンコールにおけるパラメータの受け渡しをすべて機械命令レベルで吸収したこと、

などに起因する。また (i) の特長は、SVP のマイクロ・アーキテクチャが 6.2.7で述べたように、連続主記憶読み出し機能や複数個の MAR(メモリ・アドレス・レジスタ) を装備していたために主記憶アクセスのオーバヘッドを吸収できたことによって初めて可能になったものである。

6.4 むすび

QA-2 のシステム管理プロセッサ SVP のマイクロ・アーキテクチャ並びにマクロ・アーキテクチャについて概説した。汎用計算機の機械命令と比較して SVP で採用した機械命令のレベルは高く (総命令ステップ数が約 1 / 3 になっている)、モジュール構造支援やマルチプロセスの実行環境を効率良く実現している。図 6.1に示した VAX11/750 で開発してきた QA-2 のシステム管理機能を SVP 上に移植した。

Chapter 7

QA-2 アーキテクチャの今日的意義

7.1 はじめに

本章では、QA-2 で試みられた、各種の機能に付いて、それが今日のプロセッサ設計にどのように生かされているかを考察する。QA-2 には、開発当時において一般的でない非常に先進的な機能が各種採り入れられていたが、これらの機能をまとめると以下のようになる。

- 低レベル並列処理機能
- RISC 風の操作 (命令) 記述
- ALU 連鎖機能
- プライオリティ・エンコードやビット/バイト処理演算
- 主記憶アクセスの non-blocking 機能
- グラフィックス・データの主記憶上展開
- 仮想制御記憶
- 高機能順序制御方式
- 複雑な分岐条件を設定可能な条件分岐機能
- システム制御機能とアプリケーション実行機能を別プロセッサに分離

以下では、上記のような機能が、今日のプロセッサ設計技術として、どのような評価が出来るか考察する。

7.2 低レベル並列処理機能

低レベル並列処理機能は、QA-2と同時期には、FPS (Floating Point Systems) 社のAP-120B [10] 等があるが、その後、Yale大学のFisherらによって、ELI-512 [14] が開発され、そこで提唱された「VLIW方式」という名前で定着した。その後、それを発展したMultiflow社のTRACE [12]、Cydrome社のCydra 5 [50] が商用計算機として発表されたが、一般的には定着しなかった。これは、命令語で陽に演算の並列性を示す為に、オブジェクトコードレベルでの互換性を取りにくいと言う点と、コンパイル時に並列性を検出できる様なアプリケーションはベクトル処理が出来る事が多く、高度に高速化されたベクトル・プロセッサとの住み分けが困難であった為である。

一方、同様な演算器レベルの並列処理としては、スーパースカラ方式がある。これは、ハードウェアで複数の命令を検査し、同時に実行できる命令を並列処理するものである [2] [36]。この方式では、命令セットアーキテクチャは、既存のものを使用することができ、オブジェクトコードレベルの互換性を保持することができる。このため、現在の汎用プロセッサでは、このスーパースカラ方式が一般的となっている。

これらの両方式を技術的に比較すると、

- 両方式とも、並列化率を高くするためには、コンパイラが並列に実行できる操作 (演算) を検出し、VLIW方式であれば1命令に詰め込み、スーパースカラ方式であれば、ハードウェアが同時実行可能であることを容易に認識できるように命令を並べ換える静的命令スケジューリングが必要がある。

- VLIW方式では、指示された通りに並列に動作するが、スーパースカラ方式では、ハードウェアがその通りに並列性を検出するかどうか分からない。逆に、VLIW方式では、並列度の上限がアーキテクチャ的に規定されてしまうが、スーパースカラ方式では、並列度の上限はハードウェアのインプリメントに依存するだけであるので、扱い得る並列度を増加させるのはアーキテクチャの変更を含まないと言う点で容易である。また、コンパイラが検出できなかった並列性 (実行時にならないと分からないものや、偶然的なもの) に対しても並列処理が可能となる。

- スーパースカラ方式は、ハードウェアが命令の並列実行可能性を検出しなければならないので、ハードウェアが複雑になり、また、この部分がマシンサイクルを決定づけることが多い。これに比べて、VLIW方式はこの様なハードウェアが不要であるためにインプリメントが簡単になる。

とまとめることができる。現在では、上記のようなVLIW方式の欠点を問題としない様な分野での活用が行われており、例えば、処理アルゴリズムが固定的で、大量データの均一処理を行うようなメディア処理付加プロセッサの分野では、メジャーな方式となっており [49] [56] [47]、

専用プロセッサの構成方式や、コンパイラによる静的命令スケジューリングでほとんどの並列性が検出できるような数値演算スーパーコンピュータの構成方式として、評価されている。

7.3 RISC風の操作 (命令) 記述

水平型マイクロ命令であるので当然と言えば当然であるが、各マイクロ操作の記述はシンプルであり、また、全体としてのマイクロ命令形式もシンプルなフィールド構成となっている。しかし、マイクロ命令と言っても、それ以前の特定命令セットを実現するための少量のマイクロプログラムをアセンブラ程度の開発環境で作成するのではなく、大規模なアプリケーションに近いアルゴリズムを実現する量の作成を高級言語による開発環境で行うことを考えていたため、いわゆる機械命令と呼んでもおかしくない位置付けである。これを命令セットと呼ぶと、現在のRISC命令セットの命令形式設計の思想と同一である。このような命令形式の特徴は、ハードウェアを設計する時に制御ハードウェア量を削減するのに役立ち、また、このためにデメリットとなるものはない。

7.4 ALU連鎖機能

QA-2では、データの依存関係が有ってもその依存関係を陽に指定することで1マイクロ命令中にそれらの操作を記述できるALU連鎖演算機能が有った。これによりマイクロ命令の静的な語数を削減できるだけでなく、命令パイプライン処理を行っていなかったQA-2ではレジスタの読み出しフェーズが並列処理されるため、複数のマイクロ命令で実行される場合より実行速度が向上した。これは、命令パイプライン処理での演算結果の演算器入力へのバイパス機能と同様であると言える。また、スーパースカラ方式の命令レベル並列処理を行うプロセッサであるTI社のViking SPARCチップも、一部の演算間では、演算器のカスケード処理と称して、データ依存性の有る2命令を演算器の連鎖を用いて1サイクルで実行する機能を装備しており、ALU連鎖演算機能の有効性は現在でも実証されている。

7.5 ビット/バイト処理演算

QA-2では、一般化された指定方式を持つビット操作・バイト操作の演算を装備した。これは、FFT処理における鏡像処理等の利用の他に、もっと一般的なビット操作も可能なように考えたが、データ構造として、ビットのフィールドを多用しない場合にはあまり利用されなかった。現在のプロセッサでは、データ中の“1”の個数を数えるなど普通の演算の組合せでは実現しにくい演算の実装は見受けられるが (SPARC V9)、一般的なビット操作命令は無い。また、メディア処理の専用プロセッサでは、汎用化は行われていないが、mpegデータの符号化復号

化処理やデータ圧縮処理等を実時間で行うために特殊なビット処理の高速化が必要であり、そのアルゴリズムに特化したビット処理を装備している [35]。

7.6 主記憶アクセスの non-blocking 機能

QA-2 では、主記憶アクセスの起動とアクセスデータの使用とを切り離して、アクセスの起動をかけてから、そのアクセスと関係の無い演算を実行できるようなアーキテクチャを採用して、主記憶アクセスの待ち時間を等価的に短く見せる事ができるような仕組みを導入した。すなわち、主記憶データを必要とする演算に先立て出来るだけアドレス計算とアクセスの起動をプログラムの前の方に移動するような最適化によって主記憶アクセス時間とその他の演算をオーバーラップさせ、主記憶からのデータ待ちによる演算の停止時間を減少させようとするものである。この考え方は、「ソフトウェア・プリフェッチ」と言う名称で方式として定着し、その効果に付いても報告されている [9]。

この考え方を発展させて、out-of-order 実行のスーパースカラ方式を採用したマイクロプロセッサでは、主記憶アクセス待ちとなった命令に後続する命令をデータの依存関係が無い限り先に実行するようなハードウェア制御を採り入れて、主記憶アクセス待ちによる演算の遅れを最小化しており、non-blocking cache として定着した [28]。しかし、スーパースカラ方式のみ、すなわち動的命令スケジューリングのみであると、メモリ・アクセス命令以降の命令で干渉関係が無く実行できる命令があれば実行しようとするものであるので、たとえそのメモリ・アクセス命令以前の命令で干渉関係が無いものが有っても、それらとメモリ・アクセス命令の実行順序を入れ換えて実行することはできない。したがって、充分な効果を得るためには、静的命令スケジューリングも採り入れる必要が有る。

7.7 グラフィックス・データの主記憶上展開

グラフィック処理は、QA-2 の重要な応用の 1 つと考えて開発されており、主記憶上に画素データを配置した領域を設け、主記憶制御部はビデオ信号生成器の要求により超高速バーストモード DMA 転送を行って、ディスプレイ装置に描画できる。この方式は、現在ではビットマップ・ディスプレイ方式として、広くワークステーションやパーソナル・コンピュータで採用されている方式である。

7.8 仮想制御記憶

QA-2 では、大規模なマイクロプログラミングに備えて、制御記憶の仮想化機能を装備していた。これは、主記憶をバックアップとして使用し、システム制御プロセッサの制御の下で、入れ換えを行うものであった。近年は、RISC プロセッサが主流を占めているため、プロセッサそのものの制御方式としてはマイクロプログラム制御はあまり使われていないが、メインフレーム・プロセッサでは、その命令機能が複雑であるために、マイクロプログラム制御が使われている。ここでは、保守・診断機能等の通常はあまり必要とならないマイクロコードを高価な制御記憶に常駐させるのはコスト的にも不利であるので、入れ換えを行っている。ただ、このような保守・診断時は一旦プロセッサを停止するのが普通であるので、動的な入れ換えは行っていないのが現状である。

7.9 高機能順序制御方式

QA-2 では、その前身である QA-1 上でのマイクロプログラム開発の経験から、順序制御の機能を高機能化している。これは、QA-1 ではプログラムのアルゴリズムとして判断しなければならない条件の数でマイクロプログラムのステップ数が決まってしまい、それに対して演算数が少ないと十分に並列性が生かせないと言う問題点があったためで、QA-2 では条件分岐や多方向分岐とマイクロサブルーチン・コール/リターンを任意に組み合わせる事が可能なようにして、判断しなければならない条件の数に対して必要なマイクロステップ数を少なくできるようにした。このことにより、インタプリタのように演算数に比べて判断条件の多いプログラムにおいても十分並列性を生かせる。

現在においても、このような分岐による並列度の低下は問題であるが、もっとも一般的な解決策は、分岐先をブランチヒストリで予測し、分岐先の命令も投機的実行を行って並列度を落さないと言うものである。また、命令として条件付き実行を行うものを導入し分岐そのものを削減する試みもある。しかし、これらの方法は、ブランチヒストリの予測が外れた時は余計な命令を実行することになったり、条件付き実行命令は条件が "偽" であった時は無操作の命令を実行することになるなど、効果がでない場合もある。

従って、確実な効果を期待するには、順序制御を高機能化して分岐を凝縮する QA-2 のような方向が一番であるが、QA-2 のように汎用性を追求するあまり分岐形式が多様になりすぎ、うまく活用できないような、特にコンパイラで扱い切れない程の多種類の組み合わせまで用意したことは、見直しと選択が必要である。その上で、使用頻度の多い、効果のある組み合わせは新規の命令として拡張していく必要がある。

7.10 条件分岐機能

上記の高機能順序制御方式と同様の理由で、QA-2 では 2 方向分岐の条件式に対して 8 変数までの任意の論理演算が許されている。このように 1 つ 1 つの条件毎に分岐を行うのではなく、その条件を論理変数として論理演算を行い、分岐は一度だけ行うと言う発想は、IBM の POWER アーキテクチャで実現されている condition register とその値を論理演算する命令群を装備する [68] と言う発想と非常に近いものである。ともに、条件分岐の条件生成とその論理

演算、分岐と言う操作を分けて、他の命令スケジューリングと同様にスケジューリングの対象として最適化を行い性能向上を図るものである。

発想は同様であるが、そのインプリメントはPOWERのほうが凝縮され無駄の無い設計であると言える。これは、QA-2のインプリメントは、論理演算を汎用かつ高速に行うために8変数の論理演算の真理値をRAMに持つと言う方式を採ったが、実際には真理値表でないと表現できないような複雑な演算はあまりなく、通常の論理演算の組合せで実現しているPOWER程度のインプリメントが実用的であった。

7.11 システム制御プロセッサ

QA-2では、実際のユーザーアプリケーションを走行させるQA-2本体と、システムの制御、いわゆるオペレーティングシステムの機能に相当する部分を実現するシステム制御プロセッサをハードウェア的に別プロセッサとし、機能分散システムとしている。現在、このような方式を採る計算機システムは無いが、プロセッサの価格が低下している状況で、このような用途をプロセッサ毎に限定したマルチプロセッサ構成と言うのも、十分実現性があると考えられる。欠点としては、多数のプロセッサが結合されていても、システム制御プロセッサが故障すると、システム全体のダウンとなるため、信頼性を保証する手段を考えなければならない。

Part II

VPP500 スカラプロセサ

Chapter 8

パイプライン処理と低レベル並列処理

8.1 まえがき

現在の汎用大型計算機は処理能力の向上のため、パイプライン処理を行うことが多い。このパイプライン処理も並列処理の一種であり、命令レベルや演算レベルのパイプライン処理は、低レベル並列処理と対照的である。ここでは、現在の汎用超大型商用計算機で行っているパイプライン処理について概観し、VPP500 スカラアーキテクチャ開発時のバックグラウンドについて説明する。

8.2 汎用超大型商用計算機のパイプライン処理

8.2.1 はじめに

現在の計算機において、命令の実行は、命令の取出し/命令の解読/オペランド・アドレスの計算/オペランドの取り出し/演算/結果の格納の各段階に分けられる。これらの段階を並列動作させて同時に複数の命令を処理できるようにした制御を、命令の先行制御とかパイプライン処理と言う [33]。最近の計算機では、何らかのパイプライン処理を行うのが普通である。最も単純な場合は、命令の解読以降の処理と次の命令の取り出しを並列に行うものであり、最新の汎用超大形計算機では、各段階を更に細分して並列多重度を高め、性能向上をねらったものもある [6]。ここでは、汎用超大形計算機でどのようなパイプライン処理を行っており、また、その時の問題点は何かを明らかにする。

8.2.2 命令の取り出し

命令の取り出しの処理は、それ以降の段階と異なり、命令の種類にあまり依存しない。即ち、分岐命令などで命令の実行順序が変えられる場合以外、連続したアドレスに格納されている命令を、順次取出せばよい。そこで、命令の取り出し段階と、命令の解読段階の間にキュー (命令

バッファ) を設けて、命令の取り出し処理を命令の実行とは独立に行い、命令取り出しの遅れが、命令の実行に直接影響しないようにするのが一般的である。しかし、この方式では、分岐命令を実行したり、先行する命令で命令バッファに先取りした命令を書き換えたりすると効果が無くなる。特に分岐命令は出現頻度が高く、そのために次のような改良が加えられている。

(1) 条件付分岐命令などで、分岐先アドレスが確定しても、条件が未確定で分岐するかどうか不明の期間がある。そのため、図 8.1に示すように命令バッファを複数本設け、分岐成功・不成功の両方の命令系列を先取りする [66]。

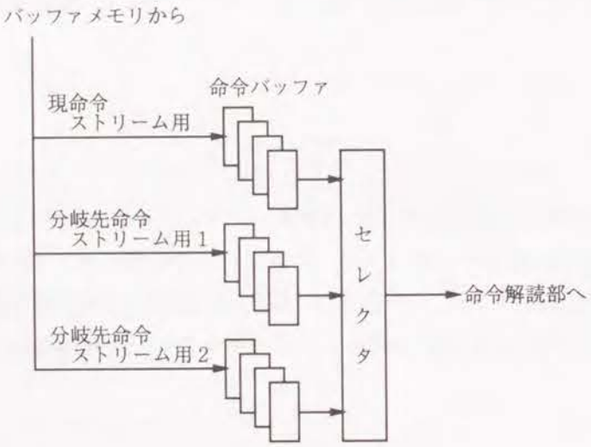


図 8.1: 複数の命令バッファによる分岐命令の高速化

(2) 図 8.2に示すような、過去に実行した分岐命令の分岐先アドレスなどを記憶しておき、それによって分岐を予測して命令の先取りを行う。(ブランチ・ヒストリ方式、分岐予測テーブル方式)[6][30]

8.2.3 命令の実行

取り出した命令を実行するためには、通常、次の 5 段階の処理をおこなう[1] [45] [63]。

1. **命令の解読：** 命令のフォーマットに従い、操作コード、オペランド指定フィールドなどを切り出す。
2. **オペランド・アドレスの計算：** ベース・レジスタ指定、インデックス・レジスタ指定により汎用レジスタを読出し、命令のディスプレースメント・フィールドの値を加えて、オペランドの実効アドレスを計算する。

[1]ここでは、IBM S370 タイプの命令セットを想定している。最近の RISC タイプの命令セットでは、これらすべての段階が必要でない場合もある。

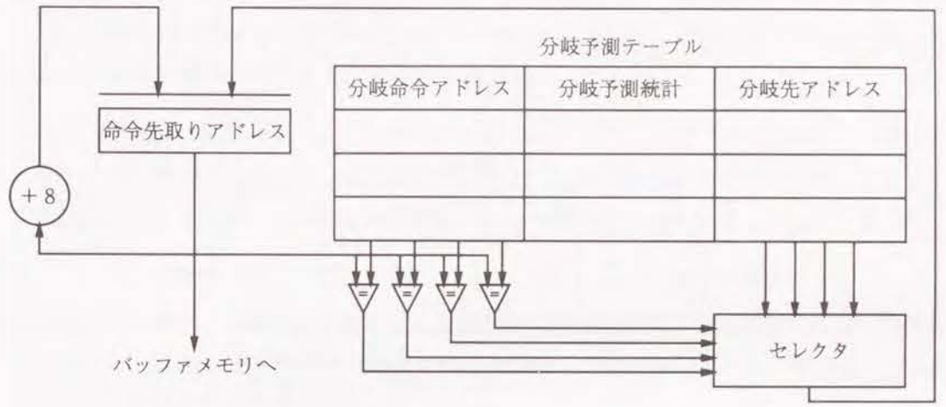


図 8.2: 分岐予測テーブルによる分岐命令の高速化

3. **オペランドの取り出し：** メモリ上のオペランドを指定された場合、実効アドレスを絶対アドレスに変換し、同時にキャッシュ・メモリにあるかどうか検索して、オペランドを取出す。通常、アドレス変換には、アドレス変換バッファ(TLB) を用いる。また、汎用レジスタ上のオペランドを指定された場合、汎用レジスタからオペランドを読出す。
4. **演算：** 操作コードに従って用意されたオペランドに対して演算を行う。一般に、演算の種類によって演算に要する時間は異なる。
5. **結果の格納：** 演算結果をレジスタやメモリに書込む。

命令の種類によっては、実効アドレスを順次更新して、(3) ~(5) を繰り返す。また、命令の最後では、割込みの有無も検査する。以上の段階を経て命令は実行されるが、高速処理を目指した計算機では、各段階を並列動作させるパイプライン処理を採用するのが普通である。パイプライン処理を概念的に示したのが、図 8.3 である。命令処理全体がパイプラインの全長に対応し、各部分が上述の (1) ~(5) の各段階に対応する、特に各部分をパイプラインのフェーズと呼ぶ (但し、(3) の段階はアドレス変換 (T) とキャッシュ・アクセス (B) の 2 つのフェーズに分かれている)。

理想的には、ある命令がパイプラインの最初のフェーズで命令の解読処理を受けて次のフェーズへ進むと、次のフェーズでのオペランド・アドレス計算処理と同時に、最初のフェーズでは、次の命令の解読処理が行われる。図 8.4のように、パイプラインの中を淀みなく命令が流れる状態が、パイプライン処理の効果を最大限に発揮した状態であり、パイプライン全体の処理時間を同じとしたとき、それをより多くの細かいフェーズに分解したほうが並列多重度が上がり、性能が向上することになる。しかし、実際の計算機では次のような要因でパイプラインに待ちが生じ、フェーズを細分化するだけでは性能向上に必ずしもつながらない [61][51] 。

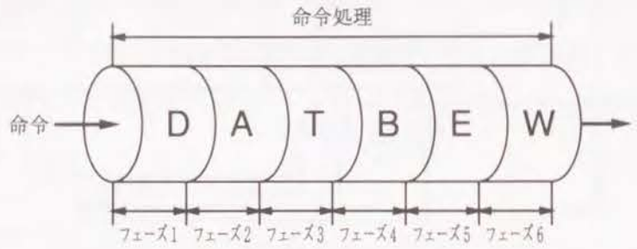


図 8.3: パイプラインの概念図

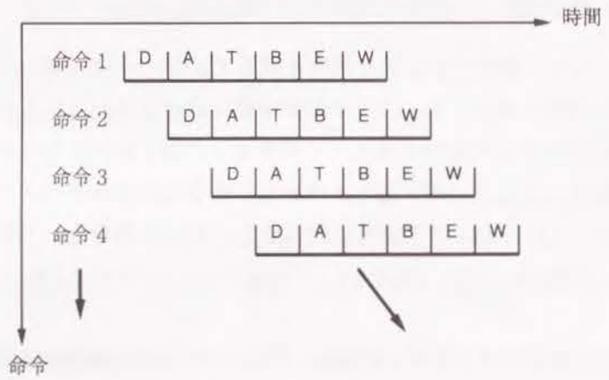


図 8.4: 理想的なパイプライン処理

命令取り出し待ち 命令バッファに処理すべき命令が取込まれていない場合で、分岐命令や先取りした命令を無効にするような制御命令を実行したときや、先取りした命令を変更するようなメモリ書込みを行ったときなどに起こる。

オペランド取り出し待ち メモリ・オペランドを取出すときに、TLB やキャッシュ・メモリに期待するデータが無い場合で、主記憶から読出すまで時間がかかり待ちが生じる。この高速化のため、キャッシュ・メモリの容量の拡大や置き換えのアルゴリズムなどを工夫している。

複雑な演算による待ち 単純な加減算に比べて複雑な浮動小数点数や 10 進数の演算は時間がかかるため、後続する演算待ちの命令は待たされる。この高速化のため、演算器をマルチ・ステージングする計算機もある [6]。

レジスタ干渉 先行する命令で書込むレジスタを、後続する命令のオペランド・アドレス計算やオペランドとして使用すると、先行する命令の書込みが終了するまで待たされる。この高速化のため、処理途中のフェーズから後続命令の他のフェーズへデータをバイパスして待ち時間を最小化する工夫が行われている。

メモリに対する書込み・読出し干渉 メモリに対する書込み・読出しにおいても、レジスタの場合と同様の待ちが存在する。メモリの場合は、書込み・読出しそれぞれがバイト単位のアドレスとデータ長を持つので、読出しデータの一部分だけが干渉する場合などがあり、レジスタの場合のように簡単にバイパスすることはできないが、条件を限ってバイパスする回路を持つ計算機もある [27]。

条件コードの設定待ち 条件分岐命令では、先行する命令が条件コードを設定する命令であると、条件コードが確定するまで分岐の成功・不成功の判定が待たされる。分岐命令の高速化とともに、この待ちの改善が図られる。

命令処理パイプラインの設計を行う場合、以上のような点に留意して、できるだけ理想的にパイプラインを淀み無く命令が流れるように、また、流れが乱れた時はできるだけ早く淀みの無い状態に戻るように各種の工夫を凝らしている。

8.3 汎用超大型商用パイプライン処理計算機の性能

8.3.1 はじめに

CPU の性能は、1 秒間に実行される命令数である MIPS 値で計られる[2]。この値は、色々な要因によりバラツキを生じる。例えば、キャッシュメモリを持たずパイプライン処理を行わな

[2] この MIPS 値は、いろいろ誤解を伴う使われ方をするが、ここでは、単に単位系と言う意味合いで使う。即ち、回転数を示す rpm と言う単位と同じ意味合いであり、計算機の性能を代表する指標という意味合いでは用いない。

い様な旧式な CPU でも、各々の命令実行速度は異なり命令の出現頻度の変化により性能バラツキを生じる。高度なパイプライン処理を行い、キャッシュメモリや TLB を装備した最新式の CPU では、これらの技術が CPU(プログラム) の振る舞いをある程度一様であると仮定した上に成り立っているので、仮定をはずれるような環境 (プログラム) ではより大きなバラツキを生じる可能性を持っている。ここでは、性能を決める要素にはどのような項目があり、どのような性格を持っているかを明らかにする。それによって CPU 性能を向上させるための指針とする。

8.3.2 性能を決める要素 — 性能の定式化 —

MIPS 値は、

$$MIPS = \frac{1}{I \times \tau} \tag{8.1}$$

I 平均命令実行サイクル数 (CPI)

τ マシンサイクル (マイクロ秒)

で表され、平均命令実行サイクル数 (I) は、

$$I = E + D + S \tag{8.2}$$

E キャッシュ上での (100%キャッシュをヒットした時の) 命令実行サイクル数

D パイプラインの干渉による遅れ

S メモリアクセスによる遅れ

に分類できる。更に、E/D/S 項は、

$$E = \sum_{i=1}^{n} C_i P_i \tag{8.3}$$

C_i 命令 i のキャッシュ上での実行サイクル数

P_i 命令 i の出現確率

$1,\ldots,n$ この CPU に定義された全命令

$$D = TKN + EEI + EGI + PRI + SFI + STI + etc \tag{8.4}$$

TKN 条件分岐の条件確定による遅れ

EEI 先行命令の結果を後続命令のオペランドとして使用するための遅れ

EGI 先行命令の結果を後続命令のベース/インデックス修飾に使用するための遅れ

PRI キャッシュアクセスの権利が取れないための遅れ

SFI 先行命令のストア・データを後続命令のオペランドに使用するための遅れ

STI 先行命令のストア・データが後続命令を書き換えるための遅れ

etc その他のパイプラインの乱れによる遅れ

$$S = \mu_O M_O + \mu_I M_I + \mu_{TLB} M_{TLB} + Se + L + M \tag{8.5}$$

μ_O オペランドのキャッシュ・ミスヒット率

M_O オペランドのキャッシュ・ミスヒット時の遅れサイクル

μ_I 命令のキャッシュ・ミスヒット率

M_I 命令のキャッシュ・ミスヒット時の遅れサイクル

μ_{TLB} TLB のミスヒット率

M_{TLB} TLB のミスヒット時の遅れサイクル

Se シリアライズによる遅れサイクル。シリアライズとは、マルチプロセッサ構成の場合に、主記憶の更新を他系のプロセッサに反映する事を保証する命令が有り、このための、ライト・バック・キャッシュであれば、更新されたキャッシュ・ブロックの主記憶への書き戻しにかかる時間、ストア・スルー・キャッシュであれば、ストアバッファが空になるまで待つ時間などが該当する。

L ロックアクセスによる遅れサイクル。これは、コンペア・アンド・スワップ等のアトミックな参照・更新の命令は、マルチプロセッサ構成時には、他系からのアクセスを排除するために特別の手順を踏む必要があり、このための遅れ時間

M ストアバッファに空きが無い場合の遅れサイクルなど。ストア・スルーのキャッシュを採用していると、ストアは毎回主記憶まで伝わる。これを高速化するために、ストアバッファを設けてストアバッファに格納する事によってストア命令の実行終了としている。このストアバッファは有限個であり、ストアが連続した時やストアバッファから主記憶への反映が遅れている時など使用可能なストアバッファが無くなり、後続するストア命令が待たされる。

に、細分される。図 8.5に、ある超大型汎用計算機[3]で各種の環境 (ワークロード) 下の、E 項/D 項/S 項を測定および机上計算した結果を示す。このデータを元に、各項の性格とこの計算機の評価を行う。最初に、各ワークロードの性格を簡単に説明する。

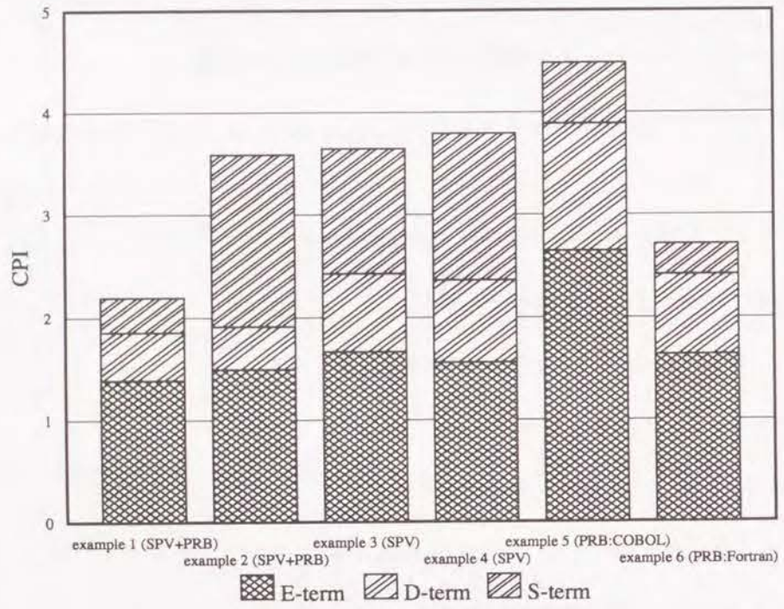


図 8.5: 平均命令実行サイクル数における E/D/S 項の構成

example1 PL/I 言語で記述された非数値処理が主な技術系アプリケーションを走行した時の、OS 部分も含む全体のワークロード。OS 構成は、example4 と同じ。実測値。

example2 example1 とは異なるアプリケーションで、example1 で作成・使用するデータベースのバックアップ処理を走行した時の、OS 部分も含む全体のワークロード。使用言語は PL/I。OS 構成は example4 と同じ。実測値。

example3 バッチ処理用の基本的な OS のスーパーバイザ部分のみのワークロード。机上計算値。

example4 example3 とは異なり、基本的な OS の上にデータベース・マネージメントシステムを搭載したスーパーバイザ部分のみのワークロード。机上計算値。

[3]命令セットアーキテクチャは、IBM S370 相当である。

example5 COBOL バッチ・ジョブのワークロードで、プロブレム状態だけを抽出した (OS 部分を取り除いた) もの。机上計算値。

example6 Fortran バッチ・ジョブのワークロードで、プロブレム状態だけを抽出した (OS 部分を取り除いた) もの。机上計算値。

次に、各項について考察を加える。

E 項 example5 以外の E 項は、ワークロードの違いにより、あまり変化しない。実行時間のかかる特権命令 (example2/3) や浮動小数点演算 (example6) も早く実行できるように設計されている。example5 で E 項が大きくなっているのは、十進演算や可変長メモリ・オペランドを操作する命令の頻度が高いためである。これは、設計の問題と言うよりも単純な操作を行う命令も、複雑な操作や大量のデータを扱う命令も同じ 1 命令として扱うこの命令セットアーキテクチャの特性と言える。

D 項 D 項は E 項に比べて約半分以下に収まっており、よく対策されていると言える。これ以上の改善を試みると、ハード量が膨大になり、また、その他の項目に悪影響を及ぼす (例えば、マシンサイクル低下) 可能性がある。

S 項 S 項は非常なバラツキを見せており、一番ワークロードの特性に左右されやすい性格を持っていると言える。もともと、example6 のような科学技術計算では、配列を順にアクセスするような場合が多くキャッシュの効果が大きい。また、スーパバイザでは、アドレスの離れたテーブルの項目をあっちこっちアクセスするため、もともと、キャッシュのヒット率が悪く、ワーストケースとして用いる事が多い。example3/4 がそのケースになるが、この場合でも、E 項を下回る程度に押えらるようにキャッシュ容量などが選ばれている。example2 は、予想を上回る大きさであるが、メモリのアクセスパターンが非常に悪いケースである。このケースそのものは、キャッシュのヒット率を下げるような外乱 (この場合は入出力処理が多くキャッシュがどんどん無効化されていた) があったためであるが、プログラムをチューニングすると、改善される場合が多い。

8.3.3 性能向上手段

以上の分析から、プロセッサ性能を上げるには、

- τ(マシンサイクル) を小さくする。
- E 項を小さくする。
- D 項を小さくする。
- S 項を小さくする。

の 4 点に分けて対策を考えれば良い。

τ(マシンサイクル) の削減

マシンサイクルを削減するには、

- 高速の素子を使用する。
- 高密度実装を行って、配線長を削減する。
- パイプライン段数を増やして、マシンサイクルを削減する。(スーパーパイプライン方式)

等が考えられる。但し、パイプライン段数の増加は、D 項の悪化につながる恐れがあるため、D 項とのトレードオフを慎重に検討しなければならない。

E 項の削減

E 項を小さくするためには、

- 命令の並列実行を行う。(スーパースカラ方式や LIW/VLIW 方式)
- 高速の演算器を用いて、複数サイクルかかる演算を削減する。
- 複数サイクルかかる演算器をパイプライン化し、後続命令の待ちを削減する。

等が考えられる。スーパースカラ方式は、非常に効果が期待できるが、そのためのハードウェア量増加や制御の論理段数の増加がマシンサイクルに影響しない様に考慮する必要がある。

D 項の削減

D 項を小さくするには、

- 分岐予測手法を採り入れる。
- バイパス回路の充実。
- 干渉の発生を低下させるためにパイプライン段数を少なくする。

等が考えられる。分岐予測方式は、プログラムの振舞いにより確率的に効果が現れるので、プログラムによっては全く効果が無い場合も存在する。また、予測がはずれた場合のペナルティが大きいと、逆効果となる場合も在り得るので、注意を要する。

また、マシンサイクルの項でも触れたが、パイプライン段数は安直に少なくするとマシンサイクルを低下させる要因となり、トレードオフの考慮が重要である。

S 項の削減

S 項を小さくするには、

- キャッシュメモリの容量を拡大して、ヒット率を上げる。(μの削減)
- 主記憶等を高速化して、ミスヒット時の遅れを小さくする。(Mの削減)
- キャッシュを階層化して、ヒット率と容量を向上する。
- ミスヒット時の遅れをコンパイラで命令スケジューリングして小さく見せる。

等が考えられる。キャッシュの容量を単純に拡大するのは、キャッシュアクセスそのものの低速化が懸念され、主記憶等の高速化は、主記憶容量が犠牲になる可能性が在る。キャッシュの階層化は上記 2 点の問題点をうまく解決する手法であるが、階層が増える事によるオーバヘッドがでない様に注意しなければならない。4 点目のコンパイラによるスケジューリング [9] であるが、命令アーキテクチャとして、メモリアクセスの非同期化を行うか、非同期プリフェッチ専用の命令を導入する事によって可能となる。

8.4 VPP500 スカラプロセッサ開発における留意点

以上のような性能向上手段を踏まえて、VPP500 スカラプロセッサの開発にあたり留意した事は次のような点である。

- これまで汎用計算機開発で蓄積して来たパイプライン処理技術のノウハウの上に、最近の RISC 技術と命令レベル並列処理を融合した、新しい命令セットアーキテクチャを策定する。
- RISC 技術の哲学は、ハードウェアとソフトウェア (特にコンパイラ) とのトレードオフの見直しであり、コンパイラの最適化技術を活用する事によって、行いたい処理を最高速に実現できるハードウェアを提供することである。即ち、コンパイラで事前にできる事に対してハードウェアを使う事無く、コンパイラが性能を向上させるために利用できるような機構にハードウェアを投資すべきである。このため、命令レベル並列処理としては、コンパイル時にコンパイラが命令の並列実行可能性を検出しているにもかかわらず、ハードウェアで実行時に再度命令の並列実行可能性を検査しなければならないスーパースカラ方式よりも、コンパイラで検出した並列実行性を直接ハードウェアに伝達できる LIW 方式を採用する。ハードウェアとしても物量削減とマシンサイクルの高速化に効果が期待できる。

- 命令レベル並列処理を導入しても、性能的に寄与するのは、E 項だけであり、同時に S 項や D 項を削減する対策を行わないと十分な効果が期待できない。これは、プロセッサのある部分を改良することによって得られる性能向上の度合に関する Amdahl の法則 [46] により明らかである。すなわち、E 項を削減しても全体的な効果としては、高々2〜3 分の 1 の効果しか得られない。上に挙げた性能向上手段の中で、命令セットアーキテクチャレベルでできる事は、S 項削減に効果があるメモリアクセスの非同期化である。この考え方は、RISC 哲学とも一致する。

- メモリアクセス以外にも複数サイクルかかりそうな浮動小数点演算も非同期化出来るようにする。

- メモリアクセスと浮動小数点演算を非同期化する事により、割り込みも対応しなければならない。ページ例外のように OS が介入して再開が必要な割り込みに対しては、これまで、命令の実行に対して厳密に同期した割り込みとする事で対応して来たが、非同期化しても再開するのに充分な情報を提供すれば、割り込みを発生する命令の実行と同期したタイミングで割り込まなくても問題無い。これによっても、ハードウェア上のクリティカル・ディレー・パスが緩和され、マシンサイクル向上に寄与することが期待できる。

- RISC タイプの命令セット (IBM S370 のようにレジスタとメモリオペランド間の演算ではなく、レジスタ間の演算のみ) にする事により、命令実行がメモリアクセスか演算かになるので、パイプライン段数を自然に少なくできる。また、分岐命令の分岐先アドレス表現として、プログラムカウンタ相対を主体とすることにより、命令を先取りした時点で分岐先命令の取り出しが行えるので、分岐命令の高速化が図れる。

以上のような点に留意して、VPP500 スカラプロセッサの開発に取り組んだ。勿論、性能向上手段として挙げたハードウェア技法も活用したのは、当然である。詳細は、章を改めて述べる。

Chapter 9

VPP500 スカラプロセサ

9.1 はじめに

VPP500 は、高性能スカラプロセサに各々ベクトル機構を付加した並列ベクトル計算機システムである。本システムの特徴は、1) 主記憶分散配置型の並列プロセサ構成により大容量主記憶および強力な主記憶データ供給能力を実現している点; 2) クロスバーネットワークを採用し例えばプロセサ間に分散配置した行列を転置する際に必要なプロセサ間一斉通信を高速化している点; 3) 個々のプロセサを高性能ベクトルプロセサとすることにより高い実効並列処理性能を実現している点; である [32][65][37]。図 9.1に示すように、ハードウェアはシステム制御を行う CP(Control processor)、演算処理を行う PE(Processing element)、これらを相互接続するクロスバーネットワークから構成される。PE はスカラプロセサ、ベクトルユニット、データ転送ユニットおよび主記憶から構成される。CP はベクトルユニットの代わりにグローバルシステムプロセサとの結合機構を有する。本章では VPP500 スカラプロセサ [40][41] の特徴について詳述する。

さて、従来の VP シリーズベクトル計算機 [64] は、M シリーズ計算機と同じスカラプロセサ・アーキテクチャを採用してきた。しかし、長い歴史を有するこのアーキテクチャは、命令実行順序の保証、割り込み発生時の PC(Program Counter) 値の保証等を規定していることから、命令の並列実行や追い越し処理などインプリメントの工夫による高速化を大きく妨げている。今後、スカラプロセサの性能を飛躍的に向上させるためには、ソフトウェアに内在する命令の並列性を最大限に引き出し、ハードウェアの並列処理に効果的に写象することのできるアーキテクチャを導入しなければならない。このような背景を基に、我々は、新しいアーキテクチャを採る VPP500 スカラプロセサを開発した。

本アーキテクチャを開発するに当たって、命令セットレベル・アーキテクチャとレジスタ・トランスファーレベル・アーキテクチャの区別を明確にした。すなわち、命令レベル並列処理の 1 つの大きな流れであるスーパースカラ方式は、レジスタ・トランスファーレベル・アーキテ

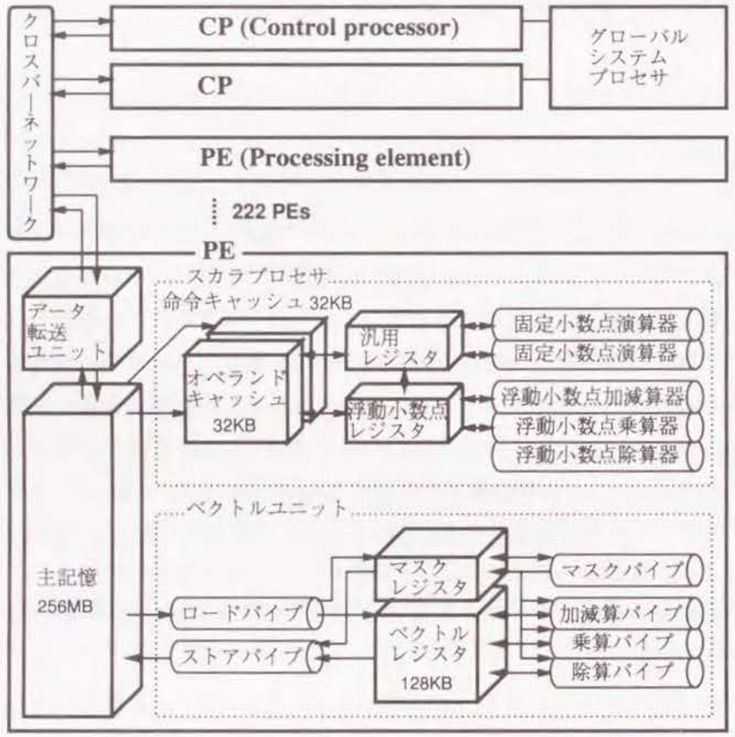


図 9.1: VPP500 システム構成

クチャの手法であり、その上位に位置する命令セットレベル・アーキテクチャには基本的に無関係である。もちろん、スーパースカラ化を適用しにくい命令セットレベル・アーキテクチャと言うのも存在し、可変長の主記憶オペランド間の演算を持つアーキテクチャや、スタック・マシンなどは、その例と言える。逆に、スーパースカラ化した時に並列度を向上させるべく分岐命令を少なくなるような工夫を施したIBM の POWER アーキテクチャのように、命令セットレベル・アーキテクチャに少しの影響を与えているものもあるが、本格的とは言えない。

そこで、本命令セットレベル・アーキテクチャは、命令/操作レベルの並列処理を最大限に活用するために必要と思われる、命令セットレベル・アーキテクチャが備えるべき仕様を余すこと無く取り込み、この仕様の下で数世代のレジスタ・トランスファーレベル・アーキテクチャを持つプロセッサ開発が、充分な自由度を持って可能なように策定された。このため、理想的な命令/操作レベル並列処理機能を持つレジスタ・トランスファーレベル・アーキテクチャを想定し、その時に並列度の向上やマシンサイクルの高速化を阻害するような要因に対して、命令セットレベル・アーキテクチャの工夫により対策できる点を盛り込みながら、まず命令セット・アーキテクチャを作成した。その後、現状のテクノロジ・レベルを前提に、VPP500 のスカラプロセサのレジスタ・トランスファーレベル・アーキテクチャを決めインプリメントを行った。したがって、命令セット・アーキテクチャとして可能なように定義はしてあるが、実際の VPP500 スカラプロセサでは、より簡単なインプリメントを行っているところもある。

本命令セットレベル・アーキテクチャの主な特徴を以下に列挙する。

1. 1 命令語当り 1 個のベクトル操作または 1〜3 個のスカラ操作を一度に発行可能とする 64 ビット長の長形式命令語 (Long Instruction Word) 方式を採用することにより、スーパスカラ方式において必要とされる並列実行可能な操作を検出するためのハードウェア機構を不要とし、また超長形式命令語 (Very Long Instruction Word) 方式の一般的な短所である命令サイズ増大を抑えたこと。命令セットレベル・アーキテクチャとしての並列度は、3 が上限となるが、その後に報告された Wall の命令/操作レベルの並列度の限界に対する研究 [67] の値からも妥当な数である。もちろん、この並列度の上限は、命令セットレベル・アーキテクチャでの上限であり、レジスタ・トランスファレベルでの並列化技法により、実マシンとしての並列度は上げることが出来る。

2. 条件分岐操作における分岐先アドレスの指定方法を PC 相対アドレスのみとすることにより、分岐先アドレスの計算および分岐先命令のプリフェッチを高速に行うインプリメントを可能としたこと。

3. 非同期操作と呼ぶ実行に複数サイクルを要する固定小数点乗算操作、浮動小数点演算操作、主記憶参照操作およびベクトル操作について、操作を並列実行する際の妨げとなる条件コード更新を必要最小限に抑えると共に、先行する非同期操作の実行完了を待たずに後続命令を発行可能とする非同期実行機構を設け、さらに、データ依存関係を崩さな

い範囲内において非同期操作の実行順序をハードウェアが変更できる規定としたこと。

4. 例外を検出した際には、実行中の操作のうち可能なものは実行を完了し、例外を検出した操作との間にデータ依存関係があるために実行不可能なものは未実行状態とした上で、例外を検出した操作の個々のアドレスが不明であってもソフトウェアが例外処理および例外処理からの復帰を正しく行うことができるよう、ハードウェアが複数の例外および未実行状態に関する十分な情報をソフトウェアに通知する機構としたこと。

次に、本アーキテクチャを最大限に利用したインプリメントの主な特徴を以下に列挙する。

1. 固定小数点演算操作および分岐操作のためのパイプラインを3段と短くし、ハードウェア量を抑えるとともに、高速な分岐先アドレス計算機構、および、次アドレスまたは分岐先アドレスから毎サイクル2命令をプリフェッチする機構を装備することにより、分岐ペナルティをほぼ0としたこと。

2. 各々独立した条件コードレジスタを有する2個の固定小数点演算器を装備し、1命令語中に記述された2個の固定小数点演算操作を毎サイクル実行可能としたこと。

3. 各々1個の浮動小数点加減算器、乗算器、除算器を装備し、特にパイプライン化された加減算器および乗算器により、1命令語中に記述された加減算操作と乗算操作の組を毎サイクル発行可能としたこと。

4. 主記憶の連続アドレスから一度に2本の32ビット汎用レジスタまたは2本の64ビット浮動小数点レジスタへデータを読み出すことができる主記憶参照機構を装備することにより、毎サイクル2個の主記憶オペランドを読み出し可能としたこと。

本章では、まず9.2においてアーキテクチャの特徴について述べ、続く9.3においてインプリメントの特徴について述べる。そこでは、随所で、レジスタ・トランスファーレベルのみで行う命令/操作レベル並列処理であるスーパースカラ方式と対比を行う。そして9.4においてLivermore14ループを用いた実測結果に基づき性能に言及する。

9.2 命令セットレベル・アーキテクチャの特徴

9.2.1 概要

VPP500スカラプロセサの命令セットアーキテクチャは、ゼロレジスタを含む33本の32ビット汎用レジスタ(レジスタ1は、ユーザ・モード用とスーパバイザ・モード用に各々1本ある)、ゼロレジスタを含む32本の64ビット浮動小数点レジスタ、内蔵TLBおよび内蔵キャッシュを有し、32ビット固定小数点演算、IEEE754準拠の32/64ビット浮動小数点演算、32ビット仮想アドレス空間、32ビット実アドレス空間およびコプロセサインタフェースを提供する汎用的

な命令セットである。また、主記憶空間とは別に、1024本分のスカラ制御レジスタ空間、8192本分のシステム制御レジスタ空間を有しており、制御レジスタを主記憶空間に割り付けた場合に生じるアドレス変換のオーバヘッドおよびTLBの消費を回避している。また、操作レベルの並列処理を含んだ命令セット・アーキテクチャであるため、既存の命令セットレベル・アーキテクチャが前提としている命令実行モデルとはすこし異なった実行モデルを想定している。

9.2.2 命令語

長形式命令語方式の採用

複数の操作を並列に実行する代表的手法に、スーパスカラ方式と長形式命令語方式がある。異なる点は、前者の場合、ハードウェアが操作間のデータ依存関係を調査し並列実行可能な操作(命令)を特定するために、相当のハードウェア量を必要とするのに対し、後者の場合、同一命令語中の操作[1]のスケジューリングを全面的にコンパイラに任せ、ハードウェアには同一命令語中の全操作を必ず同時に実行開始することだけが要求されることから、ハードウェアを軽量化できることである。このような理由から、VPP500では長形式命令語方式を採用した。さらに、ベクトル操作を含めた全命令語を固定長とすること、および、命令サイズの増大を抑えることを目的として、64ビット長の命令語中に1個のベクトル操作または1~3個のスカラ操作を記述する独自の命令語形式を採用した。おおまかに最大2個の固定小数点演算操作、最大1個の分岐操作、最大1個の固定小数点乗算操作、最大2個の浮動小数点演算操作、最大1個の主記憶参照操作を組み合わせて1命令語中に記述することが可能である。

命令語および操作の形式

図9.2に命令語の形式、図9.3に操作の形式を示す。命令語の先頭4ビットが命令語の形式を規定する。この値により、残る60ビットの領域が、20+20+20ビット長操作、20+40ビット長操作、60ビット長操作のいずれであるか、また、各操作がどのグループに属するかが決まる。グループおよび各操作の先頭3~5ビットが示すオペコードの組により、操作が一意にデコードされる。20ビット長操作は3オペランド形式を基本とし、第2オペランドとして5ビットのレジスタ番号または即値を指定可能である。40ビット長操作はオペランドとして25ビットまたは32ビットの即値を指定可能である。操作毎に、出現頻度の高い即値領域ビット長が異なることを利用し、20ビット長操作と40ビット長操作の両方にオペコードを用意する、または、いずれか一方に制限することにより、5ビットのオペコードを効率よく使用している。例えば図9.3の(*1)に示すように、乗算および論理演算に使用できる即値は25ビット即値だけである。これは一般的に、2の巾乗倍を除く乗算および論理演算は、出現頻度が極めて小さく、かつ、5ビット即値に収まる演算よりも25ビット即値を必要とする演算の出現頻度が大きいことに基づいている。(*2)に示すように、ロード/ストア操作についてもインデックス指定には25

[1] スーパースカラ方式では、命令に相当するものを長形式命令語方式では操作と呼び、同時に処理される操作の全体を命令と呼ぶことにする。

0 3	4 23	24 43	44 63
2	load,store	float add/sub/cnv	float mul/div
3	fixed op.,shift	float add/sub/cnv	float mul/div
4	fixed op.,shift	load,store,move	float add/sub/mul/div
5	fixed op.,shift	fixed op.	float add/sub/mul/div
6	fixed op.,shift	load,store,move	branch
7	fixed op.,shift	fixed op.	branch
8	load,store,move	fixed op.	
9	fixed op.,shift	fixed op.,load,store,mul	
A	float add/sub/mul/div	fixed op.,load,store,mul	
C	load,store,move,mul	call,branch,set	
D	fixed op.,shift	call,branch,set	
E	float add/sub/mul/div	call,branch,set	
F	coprocessor (vector etc.)		
B	control		

図 9.2: 命令語の形式

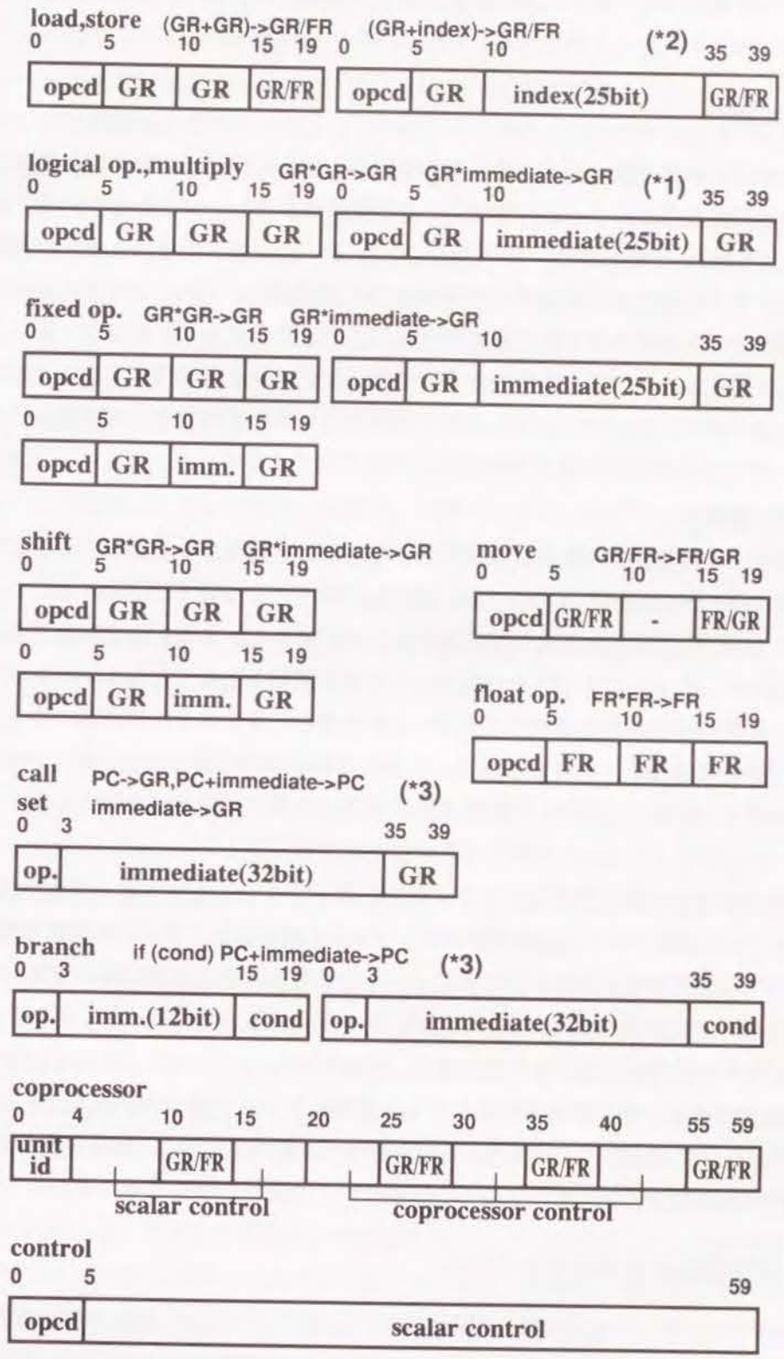


図 9.3: 操作の形式

ビット即値のみ使用可能である。(*3) に示す条件分岐操作および call 操作では、PC 相対アドレスにより分岐先アドレスを指定する。

9.2.3 同期操作

VPP500 では、乗算を除く固定小数点演算操作および分岐操作を同期操作と呼ぶ。本節では簡単のために同期操作についてのみ説明し、非同期操作を含む正確な命令実行モデルの規定については9.2.4において説明する。ハードウェアは、同一命令語に含まれる全ての同期操作の入力オペランドを同一命令語中の操作が更新する前に読み出す。即ち、同一命令語中において、複数の同期操作の入力オペランドとして同一レジスタを指定することができ、また、入力オペランドとして指定したレジスタを実行結果の格納先レジスタとして指定することができる。ただし、同一命令語中において、複数の同期操作の実行結果の格納先として同一レジスタを指定した場合、どの操作の実行結果が格納されるかは予測できない。

固定小数点演算操作

1 命令語中に 2 個の固定小数点演算操作および 1 個の分岐操作を記述することができる。例えば、操作 “add grX,0,grY” と操作 “add grY,0,grX” を 1 命令語中に記述することにより、汎用レジスタ X と Y の内容を 1 命令で交換することができる。また、文字列処理の際に必要な NULL 検出処理のために、レジスタ内容をバイト毎に比較し少なくとも 1 バイトが一致すれば条件コードが真になる操作を用意しており、1 命令語中に本操作を 2 個記述することにより一度に 8 文字分の処理を行うことができる。2 個の固定小数点演算器は、7 ビットの条件コードレジスタを各々1 組有しており、更新結果は次命令の条件分岐操作に使用することができる。

分岐操作

条件分岐操作では、常に真である 1 ビット、各々7 ビットからなる 2 組の固定小数点条件コードレジスタ、2 つの零フラグの論理和を示す 1 ビット、後述する 7 ビットの浮動小数点条件コードレジスタの合計 23 ビットの中から 1 ビットを指定し、そのまま用いるかまたは反転して用いる。例えば、比較操作 “sub grX1,grY1,gr0” と “sub grX2,grY2,gr0” を 1 命令語中に記述し、2 つの零フラグの論理和を用いることにより、2 組の汎用レジスタ X1,X2 と Y1,Y2 を一度に比較することができる。条件分岐操作および call 操作において指定する分岐先アドレスは、全て PC 相対アドレスである。この規定は、9.3.2において述べるように、分岐ペナルティの削減のために大きな効果がある。

9.2.4 非同期操作と命令実行モデル

VPP500 では、実行に複数サイクルを要することが予想される、固定小数点乗算操作、浮動小数点演算操作、主記憶参照操作およびベクトル操作を非同期操作と呼び、先行する非同期操作の実行終了を待たずに後続命令を毎サイクル発行可能とする非同期実行機構を規定している。非同期操作は、操作のキューイングまたは入力オペランドの読み出しを行う同期実行部分と、

操作の実行と結果の格納を行う非同期実行部分とに分けて実行される。9.2.3において述べた同期操作は、この同期実行部分のみから成る操作である。

命令実行モデルとしてのプロセッサは、取り出した 1 つの長形式命令語に含まれる複数の操作に対して、その同期実行部分を完了することによって命令全体の完了として PC の更新などの命令完了処理を行う。1 つの命令語に含まれる操作のうち 1 操作でもデータの依存性などで同期実行部分が完了できなければ、命令語全体の完了が待たされ、次の命令語の処理には入らない。非同期実行部分は、この命令語の取り出し・同期実行部分の実行とは独立に実行可能な時に実行される。したがって、例えばある命令語で汎用レジスタに主記憶データを読み出す操作があると、そのデータの主記憶アドレスが作成されアクセス・オペレーション・キューにエンキューされた段階でこの操作の同期部分は完了し命令も完了と見なされる。後続する命令に含まれる全ての操作の同期実行部分でこの汎用レジスタが参照 (または更新) されなければ、その命令は主記憶データがその汎用レジスタに格納されたかどうかに係わらず完了できる。また、後続する命令に含まれるどれかの操作の同期実行部分でこの汎用レジスタが参照 (または更新) される場合に、まだ主記憶データが格納されていなければ、格納されるまでこの長命令語に含まれる全ての操作の同期実行部分の実行が待たされる。

命令セット・アーキテクチャで、このような命令実行モデルを想定すると、今までのプログラミング・モデルからすこし拡張して考えることが必要になる。すなわち、今までのプログラミング・モデルでは、プログラムがどこまで実行されたかを示すプログラム・カウンタの値により、計算機システム内のプログラマブルな状態 (例えばレジスタの値とか主記憶のビットパターン) は一意に定義できた。しかし、このような新しい命令実行モデルでは、プログラム・カウンタの値のみからではレジスタなどの値が決定できない瞬間があることを認めている。もちろん、「プログラム・カウンタの値で計算機システムの内部状態が規定できない」というのは「プログラムできない」と同じ意味であるので、「プログラム・カウンタの値と、プログラムから可視な非同期実行部分の実行を規定するキューレジスタの内容から、計算機システムの内部状態が規定される」と拡張解釈する。

このようにプログラミング・モデルが拡張されると実際のユーザがプログラムする時に何らかの対応が必要になるように感じるかも知れない。しかし、この拡張により影響を受けるのはオペレーティングシステムのプログラマだけであり、一般のプログラマからは、命令セット・アーキテクチャでデータの依存性を保証しているので、拡張された事象は観測されない (割り込みが発生した時しか、このような事象は観測されない)。また、このような拡張を行うことによって、レジスタ・トランスファーレベル・アーキテクチャを考える時の自由度が増え、ハードウェア量を削減することが出来る。これに対する代償は、オペレーティングシステム内の割り込みハンドリングがすこし複雑になるだけである。

また、この命令実行モデルでは、同一命令語中に含まれる全ての同期操作および非同期操作の入力オペランド (条件分岐操作の場合には条件コードを含む) を同一命令語中の操作または後

統命令語中の操作が更新する前に読み出す。一方、同一命令語中に含まれる複数の非同期実行部分は、互いに独立に動作してよく、ハードウェアは非同期実行部分を並列に処理することができる。即ち、同一命令語中において、複数の同期操作または非同期操作の入力オペランドとして同一レジスタを指定することができ、また、入力オペランドとして指定したレジスタを実行結果の格納先レジスタとして指定することができる。ただし、同一命令語中において、複数の同期操作または非同期操作の実行結果の格納先として同一レジスタを指定した場合、どの操作の実行結果が格納されるかは予測できない。

以上のような命令実行モデルでは、コンパイラが、先行操作の非同期実行部分が終了するまで先行操作の実行結果を参照する新たな後続操作を発行しないように、操作のスケジューリングを行うことにより、非同期操作の見かけの実行サイクル数を同期実行部分に要するサイクル数のみとすることができ、非同期操作を含む命令列も最小間隔で実行することが可能となる。なお、9.4においてスケジューリングの具体例を示す。

例外検出時の基本動作

ところで、先行操作の完了を待たずに次々と操作を発行するためには、例外発生時の割込み処理に関して特別な考慮が必要となる。この命令実行モデルでは、命令列の実行中に例外を検出した場合、全ての非同期操作の実行終了を待ち合わせた後に割込みを発生し、1 個または複数個の例外を同時に報告する。この際、例外を検出した操作との間にデータ依存関係があるために実行できなかった操作は未実行操作として報告する。

まず、例外の要因が同期操作または非同期操作の同期実行部分のみに存在する場合について説明する。この時点で検出される例外は全て、未定義命令や特権例外など、命令アドレスの通知のみを必要とし、オペランドの通知を必要としないものである。また、OPC(Old PC:更新前の PC の値) によりその操作のアドレスを特定することができる。従って、ハードウェアはソフトウェアに対し、例外種別および OPC だけを通知する。

次に、例外の要因が非同期操作の非同期実行部分に存在する場合について説明する。この時点では、OPC の値が先に進んでいる可能性があり、OPC の値により操作を特定することができない。また一般に、例外を検出した操作のアドレスだけでなくオペランドの通知を必要とする。例えば、ストア操作実行時にページフォルトを検出した場合、ページインを行った後ストア操作を再実行するために、ストアデータおよびストア先アドレスの通知を必要とする。もし、後続操作によるレジスタへの上書きを止める機構を設ければ、オペランドをレジスタ中に保持することができるため、操作アドレスから操作を取り出し、レジスタ番号を元に必要なオペランドを得ることができる。しかし、後続操作の実行を止める機構は、ハードウェアを複雑化するばかりでなく、後続操作の実行が遅れる要因となる。一方、オペランドをレジスタから取り出し、他の手段により通知することができれば、後続操作によりレジスタが上書きされても構わない。この命令実行モデルでは、ハードウェアの軽量化のために主に後者の方法を採用しており、例外を検出したまたは未実行となった操作およびオペランドの多くを前述の制御レジス

タにより通知している。通知内容の詳細については、個々に後述する。

固定小数点乗算操作

固定小数点乗算操作は条件コードを更新せず、また演算時に例外を検出することもない。本規定により、条件コードに関する操作間の依存関係を無くし、操作の並列実行可能性を高めるとともに、非同期実行部分を処理するハードウェアの軽量化を図っている。固定小数点乗算操作は、同期実行部分において汎用レジスタから入力オペランドの読み出しを行い、非同期実行部分において乗算を行って汎用レジスタに対する乗算結果の格納を行う。

浮動小数点演算操作

浮動小数点演算操作に関しては、7 ビットの浮動小数点条件コードレジスタを 1 組規定しており、条件分岐に使用することができる。条件コードレジスタは、比較操作以外の演算操作が更新することはできない。本規定により、固定小数点乗算操作と同様、条件コードに関する操作間の依存関係を最小限とし、操作の並列実行可能性を高めている。また、浮動小数点演算は、一部の機能をソフトウェアがシミュレートすることにより、IEEE754 の単精度/倍精度浮動小数点演算に準拠しており、ハードウェアの軽量化を図っている。例えばハードウェアは、介入要桁あふれ例外、介入要下位桁あふれ例外、無効操作例外、零除算例外および不正確例外を検出する。このうち介入要桁あふれ/下位桁あふれ例外はマスクすることができない。ハードウェアが浮動小数点レジスタに格納した中間結果を使用して、ソフトウェアが IEEE754 準拠の桁あふれ/下位桁あふれ例外をシミュレートしている。浮動小数点演算操作は、同期実行部分において制御レジスタ空間内の FQ(Floating-point operation queue) と呼ぶキューイング機構にエンキューされる。次に非同期実行部分において、ハードウェアがレジスタ依存関係に基づき浮動小数点レジスタの読み出し、演算、演算結果の格納を行う。本命例実行モデルとしては、レジスタ依存関係が無い場合の演算順序変更を許しており、ハードウェアは FQ にエンキューされた操作を任意の順序で実行することができる。

浮動小数点演算例外検出時の動作

前述の例外を検出した際には、操作のオペコードおよびオペランドレジスタ番号が、FQ のエントリ毎に設けた例外表示領域に表示される。例外を検出した先行操作との間にレジスタ依存関係があるために実行できない後続の浮動小数点演算操作は、同様に FQ 内に未実行操作として表示される。ソフトウェアに対しては、例外処理を行った後、FQ 内において未実行となった操作を FQ の先頭から詰め直すことが要求される。ハードウェアは再構成された FQ の内容に基づき浮動小数点演算を再開する。ところで前に述べた通り、オペランドを表示する際には、レジスタ番号ではなくレジスタ内容を表示するべきである。しかし、各々64 ビットの浮動小数点数を多数格納するためには相当のハードウェア量を必要とする。ハードウェア量を抑えるためには、むしろ後続操作の実行を停止させる方法が有効であるため、浮動小数点演算例外については、レジスタ番号を表示する規定としている。

主記憶参照操作

主記憶参照操作に関しては、1命令語中に最大2個の固定小数点演算操作または最大2個の浮動小数点演算操作を記述できることに対応して、主記憶の連続アドレスから、連続する2本の汎用レジスタへの8バイトロード操作、同様に連続する2本の浮動小数点レジスタへの16バイトロード操作を規定している。主記憶参照操作は、同期実行部分において制御レジスタ空間内のAQ(Access operation queue)と呼ぶキューイング機構にエンキューされる。次に非同期実行部分において、ハードウェアがアドレス変換を行い、レジスタ間および主記憶アドレス間の依存関係に基づき主記憶参照を行う。浮動小数点演算操作同様、本命令実行モデルは、レジスタ間および主記憶アドレス間に依存関係が無い場合の主記憶参照順序の変更を許しており、例えば、先行するロード操作がキャッシュミスした場合でも、後続のロード操作がキャッシュヒットした場合には、後続のロードデータを先に使用して演算を続行することができる。即ちソフトウェアによる主記憶からキャッシュへのプリフェッチ[9]を実現可能としている。命令語列の空き部分に、プリフェッチのためのロード操作を埋め込むことにより、命令語数を増やすことなくプリフェッチを行うことができる。

主記憶参照例外検出時の動作

ページフォルト等の例外を検出すると、1) ロード操作の場合、主記憶仮想アドレス、データ長、ロードデータ格納先レジスタ番号; 2) ストア操作の場合、主記憶仮想アドレス、データ長、ストアデータ; がAQのエントリ毎に設けた例外表示領域に表示される。例外を検出した先行操作との間にデータ依存関係があるために実行できない後続の主記憶参照操作は、同様にAQ内に未実行操作として表示される。ソフトウェアに対しては、ページイン等の処理を行った後、例外を検出した操作および未実行操作を再実行することが要求される。

ベクトル操作

ベクトル操作は、ベクトルレジスタ以外に、汎用レジスタまたは浮動小数点レジスタをオペランドとする場合がある。同期実行部分において汎用レジスタまたは浮動小数点レジスタを読み出した後、ベクトル操作は、非同期実行部分を司るベクトルユニットに対してエンキューされる。ベクトル操作において指定した汎用レジスタまたは浮動小数点レジスタに関し、先行する操作との間にレジスタ依存関係が存在する場合には、ハードウェアがベクトル操作の発行を待ち合わせる。同様に、先行するベクトル操作が実行結果を汎用レジスタまたは浮動小数点レジスタに格納し、後続操作がこれを参照する場合、ハードウェアが後続操作の発行を待ち合わせる。

9.3 インプリメントの特徴

9.3.1 概要

図9.4にスカラプロセサの構成を示す。スカラプロセサはIU (Instruction control unit)、MU (Fixed-point multiply unit)、FU (Floating-point unit)、AU (Access control unit) と

呼ぶ各々独立に動作する4つのユニットから構成される。IU、MUおよびFUの実現には、ゲート遅延時間60ps(Pico second)のGaAs(Gallium arsenide)素子によるゲート数25000のLSIを5個、キャッシュを含むAUの実現には、ゲート遅延時間70ps、RAMアクセス時間1600ps、RAM容量64KbitのECL(Emitter coupled logic)素子によるLSIを13個使用している。同期操作および同期実行部分をIUが処理し、非同期実行部分をその他のユニットが処理することにより、固定小数点演算操作、固定小数点乗算操作、浮動小数点演算操作、主記憶参照操作およびベクトル操作を並列に実行する。

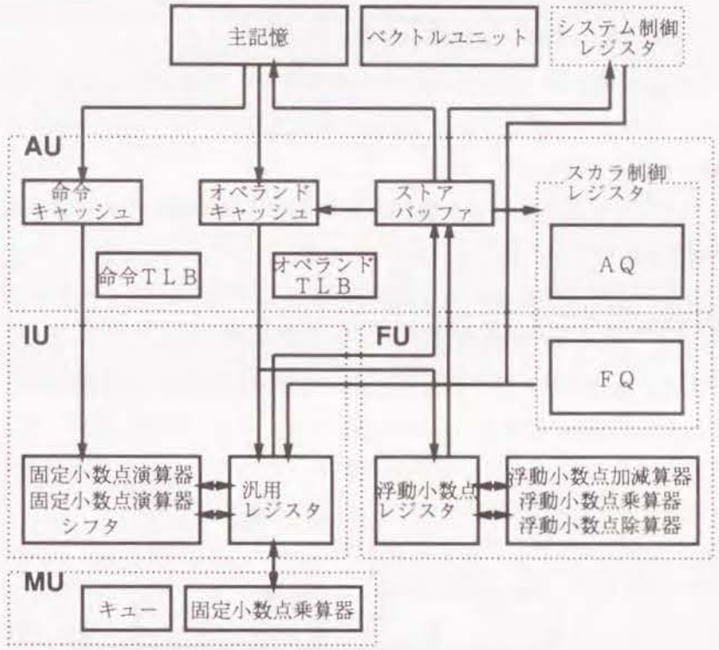


図 9.4: スカラプロセサの構成

各ユニット内およびユニット間では、先行操作と後続操作の間にデータ依存関係が存在する場合にのみインタロックが生じる。例えば先行操作が主記憶からのロード操作、後続操作が浮動小数点演算操作である場合を考える。一般にロード操作の実行には、キャッシュミス等により多くのサイクル数を要する場合がある。ロード操作の非同期実行部分はAU、浮動小数点演算操作の非同期実行部分はFUが実行する。後続の浮動小数点演算操作がオペランドとしてロード結果を使用する場合には、FUがAUの処理完了を待ち合わせる。一方、ロード結果を使用しない場合には、FUはAUの処理完了を待つことなく浮動小数点演算を開始する。即ち後者の場合、AUにおいてキャッシュミスが検出されロード操作の実行が遅延した場合においても、このためにFUにおける浮動小数点演算が遅延することはない。表9.1に、主なパイプライン・

インタロックによるペナルティ・サイクル数を示す。

操作 A の結果を操作 B が使用する場合 A → B と記す	ペナルティ サイクル数
固定小数点演算→条件分岐	≃0
浮動小数点演算→条件分岐	≃3
固定小数点演算→固定小数点演算	0
固定小数点乗算→固定小数点演算	5〜7データ依存
浮動小数点加減乗算→浮動小数点演算	3
浮動小数点除算→浮動小数点演算	4〜9データ依存
ロード→固定小数点/浮動小数点演算	2

表 9.1: 主なペナルティ・サイクル数

9.3.2 命令パイプライン

命令パイプラインは 3 ステージからなり、9.3.5において述べる命令フェッチパイプラインと密接な関係がある。図 9.5に示すように命令パイプラインは、操作のデコードおよび汎用レジスタの読み出しを行う D ステージ、演算を行う E ステージ、演算結果を汎用レジスタに格納し、条件コードおよび PC を更新する W ステージから構成される。

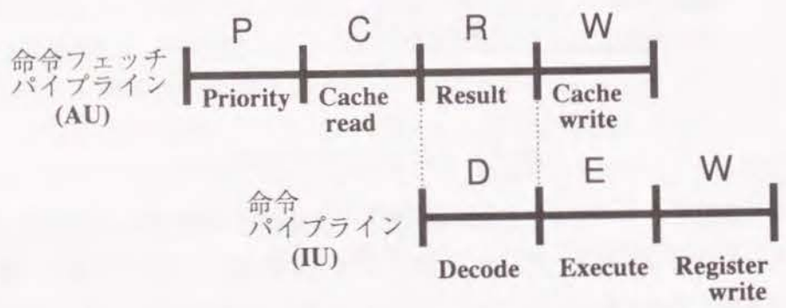


図 9.5: 命令パイプライン

命令フェッチについては、以下の方式により、分岐操作のフェッチと同時に分岐先アドレス計算を行うこと、および、分岐操作のデコードと同時に分岐方向を決定することが可能となった。分岐が連続し、プリフェッチが間に合わない場合を除き、分岐時のペナルティは 0 である。

1. 各々2 命令分の現命令バッファ、次命令バッファおよび分岐先命令バッファを装備し、128 ビット幅のデータパスを命令キャッシュとの間に設けることにより、連続する 2 命令を毎サイクルプリフェッチする。

2. 9.2.3において述べたように、条件分岐操作および call 操作における分岐先アドレスが常に PC 相対アドレスであることを利用して、PC と PC 相対アドレスを常に加算しておき、分岐操作のフェッチと同時に分岐先アドレスを得る。

3. 命令パイプラインを 3 ステージと短くし、分岐操作のデコード前に直前の固定小数点演算操作が更新する条件コードを参照可能とする。

一方、サブルーチンから戻る際に使用する jmp 操作 (レジスタ間接分岐) の場合、分岐先アドレスはレジスタ中に保持されているため、上記方式だけでは jmp 操作のフェッチと同時にアドレス計算を行うことができない。jmp 操作については、以下の方式により高速化を行った。まず、VPP500 ではサブルーチン呼び出し時に call 操作を使用し汎用レジスタ GR3 に戻りアドレスを格納するとした。次に、GR3 に格納した値をハードウェアが jmp 操作の分岐先予測アドレスとして内部バッファに保持し、GR3 を使用する jmp 操作を実行するまでに GR3 に対する書き込みが行われなければ、本内部バッファの内容を分岐先アドレスとして使用するとした。本機構により、条件分岐操作と同様、jmp 操作についても分岐先アドレスを予め知ることができる。

9.3.3 固定小数点演算機構

IU は、33 本の 32 ビット汎用レジスタ、2 個の ALU および 1 個のバレルシフタを有する。IU はパイプライン化されており、「ALU 操作またはシフト操作」と「ALU 操作」の 2 つの操作を毎サイクル同時に実行することができる。

MU は、最大 1 個の乗算操作の非同期実行を可能とするキューイング機構および乗算器からなる。32 ビット × 32 ビットの乗算器は、32 ビットまたは 64 ビットの演算結果を生成することができる。なお、MU はパイプライン化されていない。即ち、先行する乗算操作が終了するまで次の乗算操作の実行が待たされる。乗算結果を次命令の入力オペランドとして使用する場合のペナルティ・サイクル数は、乗数値の大小に依存しており、5〜7 である。

9.3.4 浮動小数点演算機構

FU は、32 本の 64 ビット浮動小数点レジスタ、最大 12 個の演算操作の非同期実行を可能とする FQ、1 個の加減算器、1 個の乗算器および 1 個の除算器からなる。加減算器では加減算操作の他、比較操作、浮動小数点-浮動小数点型変換操作、浮動小数点-固定小数点型変換操作を行う。除算器を除く加減乗算器がパイプライン化されており、「加減算」と「乗算」の 2 つの操作を毎サイクル同時に実行開始し、一方で演算結果を生成することができる。「除算」につい

ては、先行する除算操作が終了するまで次の除算操作の実行が待たされる。なお、操作間にレジスタ依存関係がない場合、加減乗算は除算を追い越して演算結果を格納することができる。

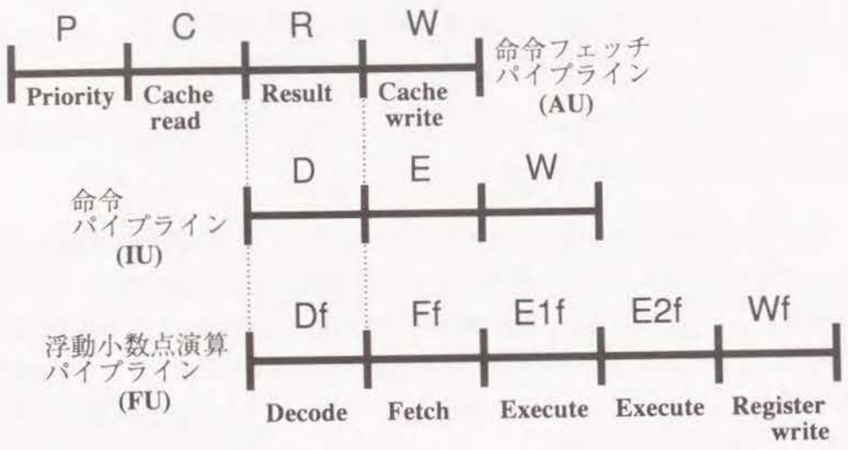


図 9.6: 浮動小数点演算パイプライン

図9.6に示すように、浮動小数点演算パイプラインは、操作をデコードするDfステージ、浮動小数点レジスタの干渉検査および読み出しを行うFfステージ、演算を行うE1fおよびE2fステージ、演算結果を浮動小数点レジスタに格納するWfステージから構成される。浮動小数点演算操作は、命令パイプラインのDステージにおいてデコードされると同時に、FQに空きがあればFQにエンキューされる。FQは、1命令語中に記述される加減算操作と乗除算操作の組を6組までキューイングすることが可能である。先行操作の実行結果を待ち合わせる必要がない場合には、同一サイクルのDfステージにおいて操作のデコードが行われる。除算はパイプライン化されておらず、E1fおよびE2fステージの代わりに複数のEfステージから構成される。但し、各Efステージを半サイクル長とすることにより除算を高速化している。加減乗算結果を次命令の入力オペランドとして使用する場合のペナルティ・サイクル数は3、除算結果を使用する場合は、被除数値の大小に依存しており、単精度演算の場合4〜6、倍精度演算の場合4〜9である。但し、レジスタ依存関係がFQ内の浮動小数点演算間に閉じている場合には、浮動小数点演算パイプラインのみがインタロックし、命令パイプラインがインタロックすることはない。即ち、FQが満杯になるか、または、浮動小数点演算以外の操作との間にレジスタ依存関係が生じた場合を除き、命令パイプラインは動き続けることができる。すなわち、浮動小数点演算操作の同期実行部分はFQへのエンキュー操作のみが含まれる。

9.3.5 主記憶参照機構

AUは、最大2個の主記憶参照操作の非同期実行を可能とするAQ、4個の8バイトストアバッファ、TLBおよびキャッシュからなる。AUはパイプライン化されており、主記憶参照操作を毎サイクル実行開始することができる。また、データキャッシュから汎用レジスタおよび浮動小数点レジスタへは、各々レジスタ2本分の幅を有するデータパスを装備しており、前に述べた、2本の汎用レジスタへの8バイトロード操作、および、2本の浮動小数点レジスタへの16バイトロード操作を他のロード操作と同じサイクル数で実行することができる。さらに、主記憶参照操作の間にデータ依存関係が無い場合、後続のロード操作が先行するロード操作を追い越すことができる機構を装備しており、9.2.4において述べたソフトウェアプリフェッチを可能とした。

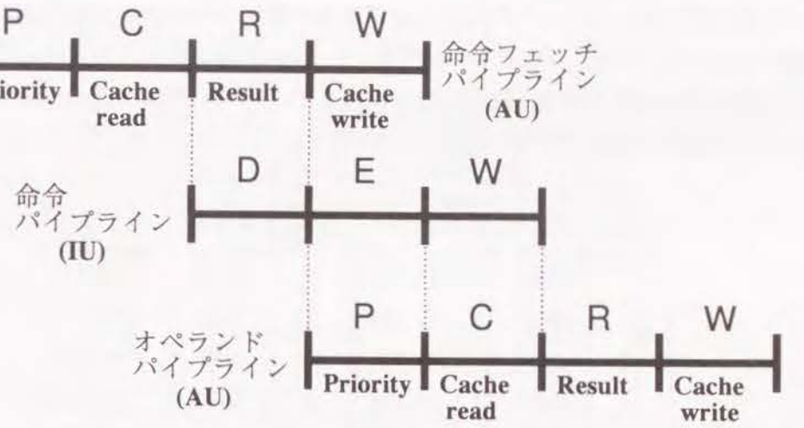


図 9.7: 命令フェッチ/オペランドパイプライン

図9.7に示すように、命令フェッチパイプラインおよびオペランドパイプラインは、キャッシュ参照権を獲得するPステージ、TLBおよびキャッシュを参照するCステージ、参照結果を得るRステージ、キャッシュに書き込みを行うWステージから構成される。Pステージでは、AUおよびIUからのキャッシュアクセス要求に対する優先順序付けを行う。Cステージでは、TLBを参照し仮想アドレスから実アドレスへの変換を行った後、キャッシュに登録されているかどうかを検査しキャッシュの読み出しを行う。Rステージでは、キャッシュヒットまたはキャッシュミスの報告、例外の報告、キャッシュへの書き込みが必要な場合の優先順序付けを行う。Wステージでは、必要な場合に応じてTLBまたはキャッシュへの書き込みを行う。キャッシュにヒットした場合、ロードデータを次命令の入力オペランドとして使用する場合のペナルティ・サイクル数は2である。

命令 TLB およびオペランド TLB にはダイレクトマップ方式を採用している。エントリ数は各々256、ページサイズは 32K バイトである。65536 個のプロセス各々に対して 4G バイトの仮想アドレス空間を提供する動的アドレス変換機構を装備しており、多重仮想アドレス空間を実現している。TLB の各エントリは、エントリ有効ビット、プロセス識別子、論理アドレス上位 9 ビット、実アドレス上位 17 ビット、ページ制御情報 (Write-protect, Read-protect, Execute-protect, Common-page, Modified-page) を保持する。

命令キャッシュおよびオペランドキャッシュにはライトスルー方式およびダイレクトマップ方式による物理キャッシュを採用している。容量は各々32K バイト、ラインサイズは 64 バイト、キャッシュミス時のペナルティは 17 サイクルである。

9.4 性能

スカラプロセサのサイクルタイムは 10ns である。最大 2 個の浮動小数点演算操作を毎サイクル発行できることから、浮動小数点演算ピーク性能は 200MFLOPS(Million floating-point operations per second) である。表9.2に、FORTRAN コンパイラを使用し、Livermore14 ループを走行した結果を示す。

Loop	演算数	MFLOPS	Loop	演算数	MFLOPS
1	2000	113.718	8	1440	52.574
2	2000	73.723	9	1700	49.312
3	2000	73.748	10	900	22.827
4	1020	34.436	11	1000	22.649
5	2000	23.584	12	1000	40.804
6	2000	23.243	13	896	8.370
7	1920	99.237	14	1650	16.357

表 9.2: Livermore14 ループの走行結果

Loop1 に関しコンパイラは、4 重ループアンローリングおよびソフトウェアパイプライニングを行っている。非同期操作であるロード (ld) および浮動小数点加算/乗算 (fadd/fmul) に関し、ロード結果は 3 サイクル後以降、演算結果は 4 サイクル後以降に使用するようにスケジューリングを行った結果、図 9.8に示す操作列が得られる。固定小数点加算 (add) により、各配列に対するロード/ストアのベースアドレスを 4 要素分ずつ更新している。各操作について、“>” の左辺は入力オペランド、右辺は格納先を示す。矢印はデータ依存関係を示す。この操作列を折り畳んだ結果、図 9.9に示す命令語列が得られる。左端の数字は図 9.2に示した命令語の形式である。この例では、38 個の操作が「64 ビット長命令*18 個=144 バイト」に収まっており、仮

にスーパスカラ方式とした場合の「32 ビット長操作*38 個=152 バイト」よりも命令サイズが小さくなっている。また、1 命令語当りの平均操作数は 2.1 個であり、命令語形式および並列実行機構が有効に使用されていると言える。

ただし、この命令語列には改良の余地がある。例えば、[*1] の各固定小数点操作を空き領域である [**1] へ移動し、[*2] のロード操作を [**2] へ移動し、さらに [*3] の浮動小数点操作と分岐操作を 1 命令にまとめることにより、ループ当りの命令語数は 16 に減少し、1 命令語当りの平均操作数は 2.4 となる。このような緻密なスケジューリングは今後の課題である。

以上に述べたように、操作の並列実行により性能向上を図るためには、ハードウェアとコンパイラの緊密な連携が必要である。特にコンパイラの果たす役割は大きく、ハードウェアの性能を十分に引き出すためには、コンパイラに対して、ハードウェアの並列実行機構を明示することが重要である。本アーキテクチャが規定する長形式命令語および非同期実行機構は、ハードウェアを高速化するための機構であると同時に、まさに、コンパイラに対して並列実行機構を明示する機構である。

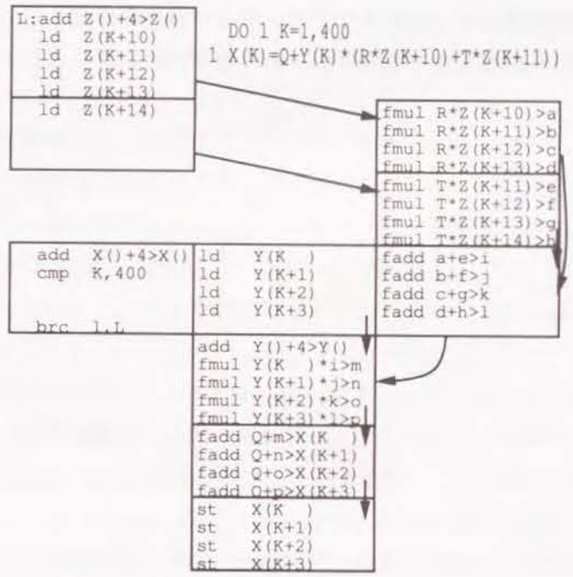


図 9.8: Loop1 の操作列

9.5 おわりに

本章では、VPP500 スカラプロセサに関し、まず、長形式命令語および非同期実行機構を大きな特徴とする命令セットレベル・アーキテクチャについて述べた。次に、本アーキテクチャ

```
7 L:add  Z()+4>Z()   add  Y()+4>Y()[*1]
2   ld   Z(K+10)     fmul Y(K   )*i>m
2   ld   Z(K+11)     fmul Y(K+1)*j>n
2   ld   Z(K+12)     fmul Y(K+2)*k>o
2   ld   Z(K+13)     fmul Y(K+3)*l>p
2   ld   Z(K+14)     fadd O+m>X(K   )   fmul R*Z(K+10)>a
3     [**1]          fadd O+n>X(K+1)    fmul R*Z(K+11)>b
3     [**1]          fadd O+o>X(K+2)    fmul R*Z(K+12)>c
2     [**2]          fadd O+p>X(K+3)    fmul R*Z(K+13)>d
2                    st   X(K   )       fmul T*Z(K+11)>e
2                    st   X(K+1)        fmul T*Z(K+12)>f
2                    st   X(K+2)        fmul T*Z(K+13)>g
2                    st   X(K+3)        fmul T*Z(K+14)>h
4 add  X()+4>X()  ld   Y(K   )         fadd a+e>i
4 cmp  K,400      ld   Y(K+1)          fadd b+f>j
4                 ld   Y(K+2)          fadd c+g>k
4                 ld   Y(K+3)[*2]      fadd d+h>l[*3]
7 brc  1,L[*3]
```

図 9.9: Loop1 の命令語列

を最大限に利用したインプリメントについて述べ、最後に、アーキテクチャおよびインプリメントが、コンパイラ技術とうまくかみ合っており、極めて有効であることを示した。今後は、分岐操作を越えて操作を移動する広域スケジューリング機能を付加したコンパイラを用い、分岐操作を多く含む命令列に関して、アーキテクチャおよびインプリメントの評価を定量的に行う予定である。

Chapter 10

結論

本論文では、10 年間と言う隔たりはあるが、命令/操作レベルで並列処理機能を持つプロセッサ 2 機種について、その設計思想と共に詳細な構成について述べた。QA-2 については、やはり実験機であり実用的な面では、機能的に取捨選択を行う余地が有った。しかし、そこで考えられた機能・方式は、いずれも興味深いものであり一度ははっきりと評価を行っておくべきものばかりであった。従って、10 年後 VPP500 スカラプロセッサの開発を開始するにあたって、QA-2 での経験が非常に有意義であった。

本論文を通して「プロセッサ・アーキテクチャはどの様にして決められるべきか」と言う問題に対して、一貫した姿勢を示したいと考えた。それは、「コストと性能と言う相反する命題に対してすべての機能・方式はそのトレードオフにより取捨選択されるべきである」という主張である。或いは、「投資と効果の評価」と言い換えれるかもしれない。即ち、ある命令を命令セットに追加するかどうかは、その命令を追加することによって必要となる追加のハードウェア量と、向上する性能が見合うかどうかである[1]。また、あるバイパスのパスをプロセッサに付加するかどうかは、そのバイパス・パスによってパイプラインの乱れが発生しなくなる確率とそのためのゲート量・配線面積やピン数とのトレードオフによって決められるべきである。このため、第 8 章としてプロセッサ性能に関する記述を含めた。より良い判断を下すためには、processor architect は、より良い performance analyst である必要がある。

さて、プロセッサ・アーキテクチャは、その性能とコストのトレードオフであることは述べたが、プロセッサだけでは、計算機システムは成り立たない。即ち、I/O 性能などのシステム・アーキテクチャが必要である。これもまた、トレードオフの上に成り立つものである。いくら高性能のプロセッサで処理時間を短縮しても、そのための初期データの入力や、その結果の出力に異常に時間が掛かってしまっては、ユーザの立場からすると処理時間の短縮は見えないものとなってしまう。VPP500 の場合、非常に高い演算性能を持っているが、それをもってしてもかなりな処理時間が掛かるアプリケーションがそのターゲット分野である。従って、システ

[1] その命令の機能に対して、現存の命令セットで代替手段が無い場合は、その機能が必要となる確率がどんなに低くても、0 で無いかぎり性能向上率は無限大である。

ム・アーキテクチャとしては、I/O 性能よりも処理性能に重点を置いたトレードオフとなっている。VPP500 をより広い分野にマッチしたものとするには、I/O 性能と処理性能のトレードオフや、スカラ処理性能とベクトル処理性能のトレードオフの見直しが必要となろう。より一般化すれば、システム・アーキテクチャを考えるに当たっては、どのようなシステムがユーザに望まれているのかを分析することによって、「投資と効果」を評価するという、もう一段階高い観点が必要であろう。processor architect と同様の表現をすれば、system architect は、より良いシステム・アーキテクチャを実現するためには、より良い system analyst でなければならない。

10 年前であれば、計算機システムの使用目的も限られたものであった。従って、そこで要求されるプロセッサ性能の基本となる機能も限られたものであった。科学技術計算/初期のバンキング・システムのような集計処理が主であって、非数値処理や、音声画像処理は、まだまだ一部で研究的に行われていたにすぎない。しかし、現在のワークステーション/ネットワーク環境では、数値処理/文字処理/データベース処理は言うに及ばず、音声/静止画像/動画像がネットワークを介してリアルタイムに遣り取りされるようになり、また、このようなマルチメディア環境がシステムとして求められている。このような背景で、計算機システム・アーキテクチャとしてプロセッサとその他のシステム構成品の関係を見直し、そこから、プロセッサ・アーキテクチャに求められる機能を見直すと言った、トップダウン的な作業も必要であり、これからの課題である。

最後に、命令セット・アーキテクチャの工学的技術的評価と産業的評価の食い違いについて考えておきたい。所謂命令セット・アーキテクチャがいかに斬新で良く考えられた技術的に素晴らしいものであっても、その命令セット・アーキテクチャが、産業界で広く受け入れられるかは別である。その大きな要因はバイナリー互換とソフト資産/ソフトウェアやミドルウェアの豊富さの問題である。また、レジスタ・トランスファーレベル・アーキテクチャとの不整合やインプリメントの巧拙により不当に評価されることもあるかもしれない。インプリメント上の問題はともかく、バイナリー互換を維持しなければならないというのは、命令セット・アーキテクチャの発展に大きな障害となっていることは確かである。また、近年は、ユーザがプログラム作成をする事が少なく、パッケージ・アプリケーションの質の高さや量の多さが、プロセッサを選ぶ基準になることが多い。これも、新しいアーキテクチャで各種のアプリケーションを集めることは難しいので、高い参入障壁となる。このために、バイナリー変換技術やエミュレーション技術が注目を集めており、これも、これからの課題である。

謝辞

この論文をまとめるに当って、京都大学の富田眞治教授、津田孝夫教授、矢島脩三教授には、大変示唆に富んだ御助言を賜わった。ここに、謝意を表したい。

10 年以上に及ぶこの研究に於いて、数多くの方々から、御助言を頂いたり、いろいろな議論に付き合って頂いて、考えをまとめるのに御協力頂いた。ここに、全ての方々のお名前を挙げる事は出来ないが、お礼を申し上げたい。

まず、計算機のアーキテクチャと言う研究の世界に私を導いて下さった萩原宏京都大学名誉教授に深甚の謝意を表したい。

京都工芸繊維大学 柴山潔教授には、在学中から現在に至るまで、御指導を賜わっている。ここに、謝意を表したい。

富士通株式会社に於いては、槌本隆光取締役、宮沢達士グローバルサーバ事業本部長代理、金田三郎グローバルサーバ事業本部主席部長、菊池伸行電算機開発部長には、本論文執筆に当って、いろいろと便宜を図って頂いた。また、追永勇次 HPC 本部技術部長には、入社以来、御指導を頂いている。ここに、謝意を表したい。

この QA-2 プロジェクトでは、中島浩氏 (現京都大学助教授)、山下博之氏 (現 NTT)、栗山和則氏 (現日立製作所)、中田登志之氏 (現日本電気)、釜田栄樹氏 (現日立製作所)、村上和彰氏 (現九州大学助教授)、笹津武司氏 (現富山富士通) をはじめとする当時の萩原研究室の方々、VPP500 スカラアーキテクチャ・プロジェクトでは、田村秀夫 HPC 本部技術部担当部長、滝内政昭ミドルウェア事業本部プロジェクト課長、上埜治彦氏、中島康彦氏をはじめとする方々と、いろいろな点に付いて議論したり、また、いろいろと助けて頂いた。ここに、謝意を表したい。

最後になったが、ここまで私がやって来れたのも、妻、純子と、二人の子ども、紗恵子、遼平のおかげである。三人に感謝の気持として、本論文を捧げたい。

参考文献

[1] 阿部, 高藤, 坂東, 平沢, 加藤, “スーパミニコン内蔵型ベクトルプロセッサの演算制御方式”, 情報処理学会論文誌, Vol.25, No.4, pp.614-621 (1984).

[2] Agerwala T., Cocke J., “High performance reduced instruction set processors”, IBM Tech. Rep., (Mar. 1987)

[3] 相磯秀夫, “メモリと計算機アーキテクチャ”, 情報処理, Vol.21, No.4, pp.304-307 (1980).

[4] 相磯秀夫, ほか, “計算機アーキテクチャ”, 岩波書店 (昭 57)

[5] 馬場, 石川, 奥田, “2 レベルマイクロプログラム制御計算機 MUNAP のアーキテクチャ”, 電子通信学会論文誌, Vol.J64-D, No.6, pp.518-525 (1981).

[6] 馬場, 渡里, 石橋, 赤木, 中村, “2 レベルのキャッシュやパイプライン処理の工夫で速度を上げた大型コンピュータ ACOS1500”, 日経エレクトロニクス,(1985.7.15)PP.233-279

[7] Barr R.G., Becker J.A., Lidinsky W.P., Tantil lo V.V., “A Research-Oriented Dynamic Microprocessor”, IEEE Trans. Comput., Vol.C-22, No.11, pp.976-985 (1973).

[8] Bongiorno V., “The CYBERPLUS Multiparallel Proces sor System”, Proc. of the Symp. on Recent Developments in Computing, Processor and Software Research for High-Energy Physics, R.Donaldson and M.N.Kreisler (eds), pp.321-331 (1984).

[9] Callahan D., Kennedy K., Porterfield A., “Software Prefetching”, Proc. Fourth Conf. on Architectural Support for Programming Languages and Operating Systems IEEE/ACM, pp.40-52 (Apr. 1991)

[10] Charlesworth W., “An approach to scientific array processing: The architecture design of the AP-120B/FPS-164 family”, IEEE Computer, Vol.14, No.12, pp.12-30 (Dec. 1981).

[11] Chu,Y., “Evolution of computer memory structure”, Proc.AFIPS National Computer Conference, pp.733-748 (1976).

[12] Colwel R. P., Nix R. P., O'Donnell J. J.,Papworth D. B., Rodman B. K., "A VLIW architecture for a trace scheduling compiler", Proc. Second Conf. on Architectural Support for Programming Languages and Operating Systems IEEE/ACM, pp.180-192 (Mar. 1987).

[13] Davidson S., Shriver B.D., "An Overview of Firmware Engineering", Computer, Vol.11, No.5, pp.21- 34 (1978).

[14] Fisher J.A., "Very Long Instruction Word Architectures and the ELI-512", IEEE Conf. Proc. of the 10th Annual Int. Symp. on Comput. Architecture, pp.140-150 (1983).

[15] 伯東株式会社 (訳), "AP-120B Processor Handbook 7259-02", Floating Point Systems Inc., p.139 (1976).

[16] 萩原宏, "マイクロプログラミング", 産業図書 (昭 52)

[17] Hagiwara H., Tomita S., Oyanagi S., Shibayama K., "A Dynamically Microprogrammable Computer with Low-Level Parallelism", IEEE Trans. Comput., Vol.C-29, No.7, pp.577-595 (1980).

[18] Hoare C.A.R.:Monitors, "an Operating System Structuring Concept", Comm. ACM, Vol.17, No.10, pp.549-557 (1974).

[19] IBM, "IBM システム/370 拡張アーキテクチャー解説書", N:SA22-7085-0

[20] 飯塚肇, "ユニバーサル・ホスト・プロセッサ", ダイナミック・アーキテクチャ, 相磯・飯塚・坂村 (編), bit・臨時増刊, Vol.12, No.10, pp.1329-1350 (1980).

[21] 池田克夫, "コンピュータ・ユーティリティの構造 — MULTICS の解剖 —", 昭晃堂 (昭 49)

[22] 和泉, 川本, 柴山, 富田, 萩原, "実時間色彩動画システムの開発", 電子通信学会論文誌, Vol.63-D, No.2, pp.1 61-168 (1980).

[23] Jones L.H., "Instruction Sequencing in Microprogrammed Computers", AFIPS Conf. Proc.of NCC, Vol.44, pp.91-98 (1975).

[24] 金井達徳, 中田登志之, 富田眞治, 萩原宏, "プログラムの解釈・実行とデータの操作・管理を分離した APL 計算機の構成", 情報処理学会・第 30 回全国大会講演論文集, 3C-1 (1985).

[25] 北村俊明, 柴山潔, 富田眞治, 萩原宏, "マイクロプログラム制御計算機 QA-1 による直接実行型高級言語計算機の構成とその問題適応化方式", 電子通信学会論文誌, Vol.J65-D, No.7, p p.882-889 (1982).

[26] 北村俊明, 中田登志之, 柴山潔, 富田眞治, 萩原宏, "ユニバーサル・ホスト計算機 QA-2 の低レベル並列処理方式", 情報処理学会論文誌, Vol.27, No.4, pp.445-453 (1986).

[27] 小高, ほか, "演算パイプラインや 3 階層記憶により高速化を図った M-680/682H の処理方式", 日経エレクトロニクス,382,PP.228-267(昭 60-11)

[28] Kroft D., "Lockup-free instruction fetch / prefetch cache organization", Proc 8th Annual Symposium on Computer Architecture, pp.81-87 (May 1981)

[29] 桑原啓治, "コンピュータ・ハードとソフトの距離を縮める高級言語マシン", 日経エレクトロニクス, pp.74-93 (Aug. 1980).

[30] Lee J.K.L., Smith A.J., "Branch prediction strategies and branch target buffer design", computer,Vol.17,No.1,JAN,1984

[31] 益田隆司, "仮想記憶の制御方式", 情報処理, Vol.21, No.4, pp.369-377 (1980).

[32] Miura K., Takamura M., Sakamoto Y., Okada S., "Overview of the Fujitsu VPP500 Supercomputer", Digest of Papers, COMPCON Spring 93, 1993.

[33] 元岡達編, "計算機システム技術", オーム社 (昭 48)

[34] 元岡達編, "VLSI コンピュータ 1", 岩波書店 (昭 59)

[35] Moussouris J.(MicroUnity), "Architecture of a Broadband Mediaprocessor", 8th Microprocessor Forum 95, (Oct. 1995).

[36] Murakami K., Kuga M., Irie N., Tomita S., "SIMP (Single Instruction stream / Multiple instruction Pipelining) : A Novel High-Speed Single Processor Architecture", Proc. 16th Int. Symp. Computer Architecture, pp.78-85 (1989)

[37] Nakanishi M., Ina H., Miura K., "A High Performance Linear Equation Solver on the VPP500 Parallel Supercomputer", Proc. Supercomputing '94, pp.803-810 (Nov. 1994)

[38] 中西直之, "最近のコンピュータ技術とバンガード・システム", インターフェース, Vol.6, No.3, pp.166-173 (1980).

[39] 中瀬、日高、西村、宮崎、野口、鷲島, "高速浮動小数点演算機能を持つユニット・コンピュータ・MC のアーキテクチャ", 情報処理学会・研究会資料, Vol.CA-85, No.15, pp.1-8 (1985).

[40] Nakashima Y., Kitamura T., Tamura H., Takiuchi M., Miura K., "Scalar Processor of the VPP500 Parallel Supercomputer", ACM Int. Conf. of Supercomputing '95, (1995).

[41] 中島康彦, 北村俊明, 田村秀夫, 滝内政昭, "VPP500 スカラプロセッサの特徴", 情報処理学会・研究会報告, 94-ARC-104-17, pp.129-136 (1994).

[42] 中田登志之, 北村俊明, 柴山潔, 富田眞治, 萩原宏, "低レベル並列処理機能を有するマイクロプログラム制御計算機 QA-2 の開発", 情報処理学会「アーキテクチャワークショップインジャパン'84」シンポジウム講演論文集 (1984).

[43] 中田登志之, 北村俊明, 柴山潔, 富田眞治, 萩原宏, "マイクロプログラム制御計算機 QA-2 のシステム管理プロセッサ", 情報処理学会論文誌, Vol.27, No.3, pp.356-364 (1986).

[44] 中田登志之, 柴山潔, 富田眞治, 萩原宏, "QA-2 を用いた LISP システムの作成", 情報処理学会・第 30 回全国大会講演論文集, 7C-3 (1985).

[45] 二宮昭一編, "新電子計算機概論 - 計算機システムと情報処理 -", オーム社 (昭 60)

[46] Patterson D. A., Hennessy J. L., "Computer Architecture : A Quantiative Approach", Morgan Kaufmann (1990) 邦訳 : 富田眞治, 村上和彰, 新實治男, "コンピュータ・アーキテクチャ ー 設計・実現・評価の定量的アプローチ" 日経 BP 社 (1992)

[47] Pechanek G.(IBM Microelectronics), "M.F.A.S.T:A Hioghly Parallel, Scalable, Single-Chip DSP", 8th Microprocessor Forum 95, (Oct. 1995).

[48] Plummer W.W., "Asynchronous Arbiter", IEEE Trans. Computer, Vol.C-21, No.1, pp.37-42 (1972).

[49] Purcell S.(Chromatic), "A VLIW and SIMD Vector Processor for PC Multimedia", 8th Microprocessor Forum 95, (Oct. 1995).

[50] Rau B.R., Yen D.W.L., Yen W., Towle R.A., "The Cydra 5 departmental supercomputer: Design philosophies, decisions, and trade-offs", IEEE Computer, Vol.22, No.1, PP.12-34 (Jan. 1989).

[51] Rymarczyk J.W., "Coding guidelines for pipelined processors", Sym. on Arch. Support for Prog. Lang. and Op. Sys.,ACM,MAR,1982,PP.12-19

[52] 柴山潔, 中田登志之, 北村俊明, 富田眞治, 萩原宏, "ユニバーサル・ホスト計算機 QA-1 による各種高級言語プロセッサのエミュレーション", 電子通信学会論文誌, Vol.J65-D, No.11, pp.1374-1381 (1982).

[53] 柴山潔, 北村俊明, 中田登志之, 富田眞治, 萩原宏, "ユニバーサル・ホスト計算機 QA-2 の高機能順序制御方式", 情報処理学会論文誌, Vol.27, No.10, pp.960-969 (1986) .

[54] 柴山潔, 中田登志之, 富田眞治, 萩原宏, "ユニバーサル・ホスト計算機 QA-2 による Prolog の処理方式について", 情報処理学会・第 30 回全国大会講演論文集, 7C-4 (1985).

[55] 柴山潔, 富田眞治, 萩原宏, "ユニバーサル・ホスト計算機 QA-2 による逐次型 Prolog マシンのエミュレーション", 電子通信学会電子計算機研究会, EC85-52, pp.1-12 (1985).

[56] Slavenburg G.(Philips Semiconductors), "The TriMedia VLIW-Based PCI Multimedia Processor", 8th Microprocessor Forum 95, (Oct. 1995)

[57] 武井欣二, "仮想記憶方式", 情報処理, Vol.21, No.4, pp.358-368 (1980).

[58] 田中善一郎, "LSI 技術の後押しで身近になった連想処理", 日経エレクトロニクス, pp.102-133 (Oct. 1980).

[59] Tomita S., Shibayama K., Kitamura T., Nakata T., Hagiwara H., "A User-Microprogrammable Local Host Computer with Low-Level Parallelism", IEEE Conf. Proc. of the 10th Annual Int. Symp. on Comput. Architecture, pp.151-157 (1983).

[60] 富田眞治, "処理装置の構成", VLSI コンピュータ, 元岡達 (編), 岩波書店, pp.12-172 (1984).

[61] 富田眞治, "並列計算機構成論", 昭晃堂 (昭 61)

[62] Tomita S., Shibayama K., Nakata T., Yuasa S., Hagiwara H., "A Computer with Low-Level Parallelism - Its Applications to 3-D Graphics and Prolog/Lisp Machines -", IEEE Conf. Proc. of the 13th Annual Int. Symp. on Comput. Architecture, pp.280-289 (1986).

[63] 槌本, ほか, "CPU を 1 ボードに実装した大型コンピュータ M-780", 日経エレクトロニクス,396,PP.179-209(昭 61-06)

[64] Uchida N., Hirai M., Yoshida M., Hotta K., "Fujitsu VP2000 Series", Digest of Papers, COMPCON Spring 90, pp.4-11, (Feb. 1990)

[65] Utsumi T., Ikeda M., Takamura M., "Architecture of the VPP500 Parallel Supercomputer", Proc. Supercomputing '94, pp.478-487 (Nov. 1994)

[66] 海野三郎, "IBM3033 プロセサの内部設計とパフォーマンス", 日経エレクトロニクス別冊「コンピュータ」,(1978.11.20)PP.85-96

[67] Wall D. W., "Limits of instruction-level parallelism", Proc. Fourth Conf. on Architectural Support for Programming Languages and Operating Systems IEEE/ACM, pp.248-259 (Apr. 1991)

[68] Weiss S., Smith J. E., "POWER and PowerPC" Morgan Kaufmann (1994)

[69] 山田博, "コンピューターアーキテクチャ", 産業図書 (昭 51)

[70] 山岡, 山本, 西山, 和田, "汎用大型計算機 M-680H/682H における十進演算命令の高速化処理方式", 第 31 回情報処理学会全国大会,1D-10,(昭 60)

[71] 安井裕, "LISP マシン", 情報処理, Vol.23, No.8, pp.757-772 (1982).

[72] 湯浅真治, 玄光均, 中田登志之, 富田眞治, 萩原宏, "マイクロプログラム制御計算機 QA-2 による実時間色彩動画システムの開発", 情報処理学会・第 30 回全国大会講演論文集, 4J-90 (1985).

[73] 湯浅真治, 中田登志之, 新實治男, 富田眞治, 萩原宏, "低レベル並列処理計算機による 3 次元色彩図形表示処理", 情報処理学会論文誌, Vol.27, No.6 (1986).

著者発表論文

論文

1. 北村俊明, 柴山潔, 富田眞治, 萩原宏, "マイクロプログラム制御計算機 QA-1 による直接実行型高級言語計算機の構成とその問題適応化方式", 電子通信学会論文誌, Vol.J65-D, No.7, p p.882-889 (1982).
2. 北村俊明, 中田登志之, 柴山潔, 富田眞治, 萩原宏, "ユニバーサル・ホスト計算機 QA-2 の低レベル並列処理方式", 情報処理学会論文誌, Vol.27, No.4, pp.445-453 (1986).
3. 柴山潔, 中田登志之, 北村俊明, 富田眞治, 萩原宏, "ユニバーサル・ホスト計算機 QA-1 による各種高級言語プロセッサのエミュレーション", 電子通信学会論文誌, Vol.J65-D, No.11, pp.1374-1381 (1982).
4. 柴山潔, 北村俊明, 中田登志之, 富田眞治, 萩原宏, "ユニバーサル・ホスト計算機 QA-2 の高機能順序制御方式", 情報処理学会論文誌, Vol.27, No.10, pp.960-969 (1986).
5. 中田登志之, 北村俊明, 柴山潔, 富田眞治, 萩原宏, "マイクロプログラム制御計算機 QA-2 のシステム管理プロセッサ", 情報処理学会論文誌, Vol.27, No.3, pp.356-364 (1986).

国際会議

1. Shibayama K., Tomita S., Hagiwara H., Yamazaki K., Kitamura T., "Performance Evaluation and Improvement of a Dynamically Microprogrammable Computer with low-Level Parallelism",Infomation Processing 80 (Proc. IFIP Congress 80, Melbourne, Australia), edited by S.H.Lavington, North-Holland, Amsterdam, Netherlands, pp.181-186, (Oct. 1980).
2. Tomita S., Shibayama K., Kitamura T., Hagiwara H., "Performance Evaluation and Improvement of a Dynamically Microprgrammable Computer with low-level Parallelism ", Proc. MICRO 13 (Colorado Springs, Colorado), ACM and IEEE, pp.79-89, (DEC. 1980).

3. Tomita S., Shibayama K., Kitamura T., Nakata T., Hagiwara H., "A High-Performance, Local Host Computer with Low-level Parallelism", Proc. Int. Symp. Appl. Math. and Inf. Sci. (Kyoto, Japan), Pt.6, Kyoto Univ., pp.9-18, (Mar. 1982).

4. Tomita S., Shibayama K., Kitamura T., Nakata T., Hagiwara H., "A User-Microprogrammable Local Host Computer with Low-Level Parallelism", IEEE Conf. Proc. of the 10th Annual Int. Symp. on Comput. Architecture, pp.151-157 (1983).

5. Miyoshi H., Fukuda M., Iwamiya T., Nakamura T., Tuchiya M., Yoshida,M., Yamamoto K., Yamamoto Y., Ogawa S., Matsuo Y., Yamane T., Takamura M., Ikeda M., Okada S., Sakamoto Y., Kitamura T., Hatama H., Kishimoto M., "Development and Archievement of NAL Numerical Wind Tunnel (NWT) for CFD Computations", Proc. Supercomputing '94, (Washington D.C.), ACM and IEEE, pp.685-692, (Nov. 1994).

6. Nakashima Y., Kitamura T., Tamura H., Takiuchi M., Miura K., "Scalar Processor of the VPP500 Parallel Supercomputer", ACM Int. Conf. of Supercomputing '95, (1995).

口頭発表

1. 北村俊明, 柴山潔, 富田眞治, 萩原宏, "QA-1 のファームウェアによる手続き向き言語の処理について" 情報処理学会・第 20 回全国大会 (東京, 日本), No.2B-6, pp.27-28, (July 1979)

2. 北村俊明, 柴山潔, 富田眞治, 萩原宏, "ファームウェアによる直接実行型高級言語計算機について" 電気関連学会関西支部大会 (京都, 日本), Pt.G, No.G6-14, PP.211, (Nov. 1979)

3. 北村俊明, 柴山潔, 富田眞治, 萩原宏, "QA-1 のファームウェアによる会話型手続き向き言語の処理システム", 電子通信学会・研究会報告, Vol.80, No.20, EC80-10, pp.45-56 (May. 1980)

4. 北村俊明, 木下耕二, 十山圭介, 柴山潔, 富田眞治, 萩原宏, "ダイナミック・マイクロプログラム制御計算機 QA-1 による高級言語処理方式について", 情報処理学会・第 21 回全国大会 (東京, 日本), No.5J-6, pp.147-148, (May 1980)

5. 柴山潔, 富田眞治, 萩原宏, 山崎勝弘, 北村俊明, 栗山和則, 中島浩, 新実治男, 山下博之, 中田登志之, 藤井誠, "低レベル並列処理機能を備えたダイナミック・マイクロプログラム制御計算機 QA-2 のアーキテクチャ", 電子通信学会・研究会報告, Vol.79, No.223, EC79-63, pp.31-40 (Jan. 1980)

6. 山下博之, 中田登志之, 藤井誠, 栗山和則, 中島浩, 新実治男, 北村俊明, 山崎勝弘, 柴山潔, 富田眞治, 萩原宏, "低レベル並列処理機能を備えたダイナミック・マイクロプログラム制御計算機：QA-2 (1) – アーキテクチャの概要、演算制御 –", 情報処理学会・第 21 回全国大会 (東京, 日本), No.3J-10, pp.111-112, (May 1980)

7. 柴山潔, 北村俊明, 富田眞治, 萩原宏, "QA-1 による各種高級言語計算機のエミュレーション" 電子通信学会情報システム部門全国大会 (金沢, 日本), Pt.2, No.S4-2, pp.404-405, (Apr. 1981)

8. 釜田栄樹, 北村俊明, 柴山潔, 富田眞治, 萩原宏, "汎用エミュレーション指向マイクロプログラム制御計算機における ALU-レジスタレベル並列処理について" 情報処理学会・第 23 回全国大会 (東京, 日本), No.1E-1, pp.51-52, (Oct. 1981)

9. 中田登志之, 北村俊明, 柴山潔, 富田眞治, 萩原宏, "低レベル並列処理機能を有するマイクロプログラム制御計算機 QA-2 の開発", 情報処理学会 「アーキテクチャワークショップインジャパン'84」 シンポジウム講演論文集 (1984).

10. 中島康彦, 北村俊明, 田村秀夫, 滝内政昭, "VPP500 スカラプロセッサの特徴", 情報処理学会・研究会報告, 94-ARC-104-17, pp.129-136 (1994).